

AD-A188 618

RESEARCH ON AUTOMATIC VERIFICATION OF FINITE-STATE
CONCURRENT SYSTEMS. (U) CARNEGIE-MELLON UNIV PITTSBURGH
PA DEPT OF COMPUTER SCIENCE. R E BRYANT ET AL. DEC 87

1/1

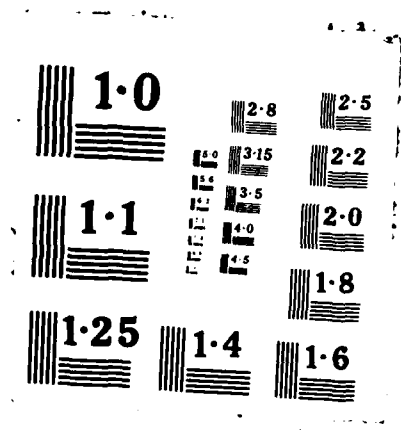
UNCLASSIFIED

CMU-CS-87-105 AFMIL-TR-87-1166

F/O 9/1

NL





AD-A188 618

DTIC ACCESSION NUMBER

LEVEL

PHOTOGRAPH THIS SHEET

INVENTORY

AFWAL-TR-87-1166

DOCUMENT IDENTIFICATION

DEC 1987

This document has been approved
for public release and sale in
distribution unlimited.

DISTRIBUTION STATEMENT

ACCESSION FOR

NTIS GRA&I ☒

DTIC TAB ☐

UNANNOUNCED ☐

JUSTIFICATION

BY

DISTRIBUTION /

AVAILABILITY CODES

DIST

AVAIL AND/OR SPECIAL

A-1

DISTRIBUTION STAMP

QUALITY
INSPECTED
2

DTIC
ELECTE
FEB 09 1988
S E D

DATE ACCESSIONED

DATE RETURNED

28 2 05 102

DATE RECEIVED IN DTIC

REGISTERED OR CERTIFIED NO.

PHOTOGRAPH THIS SHEET AND RETURN TO DTIC-DDAC

AD-A188 618

AFWAL-TR-87-1166

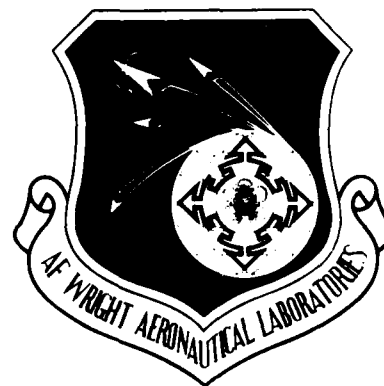
RESEARCH ON AUTOMATIC VERIFICATION OF FINITE-STATE
CONCURRENT SYSTEMS

E.M. Clarke and O. Grumberg

Carnegie-Mellon University
Computer Science Department
Pittsburgh, PA 15213-3890

December 1987

Interim



Approved for Public Release; Distribution is Unlimited

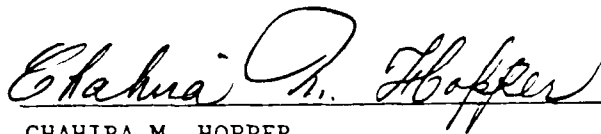
AVIONICS LABORATORY
AIR FORCE WRIGHT AERONAUTICAL LABORATORIES
AIR FORCE SYSTEMS COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OHIO 45433-6543

NOTICE

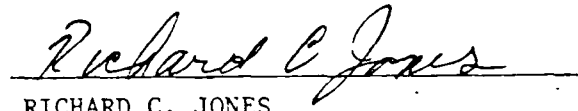
When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely Government-related procurement, the United States Government incurs no responsibility or any obligation whatsoever. The fact that the Government may have formulated or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication, or otherwise in any manner construed, as licensing the holder, or any other person or corporation; or as conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

This report has been reviewed by the Office of Public Affairs (ASD/PA) and is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.

This technical report has been reviewed and is approved for publication.



CHAHIRA M. HOPPER
Project Engineer



RICHARD C. JONES
Ch, Advanced Systems Research Gp
Information Processing Technology Br

FOR THE COMMANDER



EDWARD L. GLIATTI
Ch, Information Processing Technology Br
Systems Avionics Div

If your address has changed, if you wish to be removed from our mailing list, or if the addressee is no longer employed by your organization please notify AFWAL/AAAT, Wright-Patterson AFB, OH 45433-6543 to help us maintain a current mailing list.

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

1a REPORT SECURITY CLASSIFICATION Unclassified			1b RESTRICTIVE MARKINGS		
2a SECURITY CLASSIFICATION AUTHORITY			3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.		
2b DECLASSIFICATION/DOWNGRADING SCHEDULE					
4 PERFORMING ORGANIZATION REPORT NUMBER(S) CMU-CS-87-105			5 MONITORING ORGANIZATION REPORT NUMBER(S) AFWAL-TR-87-1166		
6a NAME OF PERFORMING ORGANIZATION Carnegie-Mellon University		6b OFFICE SYMBOL (If applicable)		7a NAME OF MONITORING ORGANIZATION Air Force Wright Aeronautical Laboratories AFWAL/AAAT-3	
6c ADDRESS (City, State, and ZIP Code) Computer Science Dept Pittsburgh PA 15213-3890			7b ADDRESS (City, State, and ZIP Code) Wright-Patterson AFB OH 45433-6543		
8a NAME OF FUNDING/SPONSORING ORGANIZATION		8b OFFICE SYMBOL (If applicable)		9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F33615-84-K-1520	
8c ADDRESS (City, State, and ZIP Code)			10 SOURCE OF FUNDING NUMBERS		
			PROGRAM ELEMENT NO 61101E	PROJECT NO 4976	TASK NO 00
11 TITLE (Include Security Classification) Research On Automatic Verification Of Finite-State Concurrent Systems					
12 PERSONAL AUTHOR(S) E. M. Clarke, O. Grumberg					
13a TYPE OF REPORT Interim		13b TIME COVERED FROM _____ TO _____		14 DATE OF REPORT (Year, Month, Day) 1987 December	
15 PAGE COUNT 31					
16 SUPPLEMENTARY NOTATION					
17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD	GROUP	SUB-GROUP			
19 ABSTRACT (Continue on reverse if necessary and identify by block number)					
20 DISTRIBUTION/AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT <input type="checkbox"/> DTIC USERS					
21 ABSTRACT SECURITY CLASSIFICATION					
22a NAME OF RESPONSIBLE INDIVIDUAL Chahira M. Hopper			22b TELEPHONE (Include Area Code) (513) 255-7865		22c OFFICE SYMBOL AFWAL/AAAT-3

Table of Contents

1. Introduction	1
2. Computation Tree Logics	3
3. The CTL Model Checking Algorithm	5
4. Fairness Constraints	7
5. An Example	9
6. Other Approaches	13
7. Applications	16
8. Conclusion	19

List of Figures

Figure 5-1:	Two process mutual exclusion program.	11
Figure 5-2:	Transcript of model checker execution (without fairness constraint).	12
Figure 5-3:	Transcript of model checker execution (with fairness constraint).	12
Figure 7-1:	Algorithm for Constructing Kripke Structure From Circuit	17
Figure 7-2:	Kripke structure for unstable configuration of AND gate.	18

Research On Automatic Verification Of Finite-State Concurrent Systems

E. M. Clarke and O. Grumberg
Carnegie Mellon University, Pittsburgh

1. Introduction

Temporal logics were first developed by philosophers for reasoning about the ordering of events in time without introducing time explicitly [Hughes & Creswell 77]. Although a number of different temporal logics have been studied, most have an operator like $G(f)$ that is true in the present if f is always true in the future (i.e., if f is globally true). To assert that two events e_1 and e_2 never occur at the same time, one would write $G(\neg e_1 \vee \neg e_2)$. Temporal logics are often classified according to whether time is assumed to have a *linear* or a *branching* structure. This classification may occasionally be misleading since some temporal logics combine both linear-time and branching-time operators. Instead, we will adopt the approach used in [Emerson & Halpern 83] that permits both types of logics to be treated within a single semantical framework. In this paper the meaning of a temporal logic formula will always be determined with respect to a labelled state transition graph; for historical reasons such structures are called *Kripke models* [Hughes & Creswell 77].

Pnueli was apparently the first person to use temporal logic for specifying and verifying concurrent programs [Pnueli 77]. His approach involved proving desired properties of the program under consideration from a set of program axioms that described the behavior of the individual statements in the program. Proofs were usually constructed by hand, and this task was in general quite tedious. Since many concurrent programs can be viewed as communicating finite state machines, there was a strong possibility that at least some of these programs could be automatically verified. The first verification technique to exploit this observation was the *CTL model checking procedure* developed by Clarke and Emerson in [Clarke & Emerson 81]. Their algorithm was polynomial in both the size of the model determined by the program under consideration and in the length of its specification in temporal logic. They also showed how *fairness* [Gabbay et al 80] could be handled without changing the complexity of their algorithm. Handling fairness was an important step since the correctness of many concurrent

This research was partially supported by NSF Grant MCS-82-16706. The second author, O. Grumberg, is currently on leave from Technion, Haifa and is partially supported by a Weinmann postdoctoral fellowship.

algorithms depends critically on some assumption of this type; for example, absence of starvation in a mutual exclusion algorithm may depend on the assumption that each process makes progress infinitely often.

At roughly the same time Quielle and Sifakis [Quielle & Sifakis 81] gave a model checking algorithm for a similar branching-time logic, but they did not analyze its complexity or show how to handle an interesting notion of fairness. Later Clarke, Emerson, and Sistla [Clarke et al 86a] devised an improved algorithm that was linear in the product of the length of the formula and in the size of the global state graph. Sistla and Clarke [Sistla & Clarke 86] analyzed the model checking problem for a variety of other temporal logics and showed, in particular, that for linear temporal logic the problem was PSPACE complete.

A number of papers have shown how the temporal logic model checking procedure can be used for verifying network protocols and sequential circuits ([Clarke et al 86a], [Mishra & Clarke 85], [Browne et al 86], [Dill & Clarke 86], [Browne et al 85], [Browne & Clarke 86], [Browne et al 86b]). In the case of sequential circuits two approaches have been developed for obtaining state transition graphs to analyze. The first approach extracts a state graph directly from the circuit under an appropriate timing model of circuit behavior. The second approach obtains a state transition graph by compilation from a high level representation of the circuit in a Pascal-like programming language. In practice the model checking procedure is able to check state transition graphs at a rate of 100 states per second for formulas of reasonable length. It has been used successfully to find previously unknown errors in published designs of asynchronous circuits.

Alternative approaches have been proposed by a number of other researchers. The approach used by Kurshan [Kurshan 86] involves checking *inclusion* between two automata on infinite tapes. The first machine represents the system that is being verified; the second represents its specification. Automata on infinite tapes are used in order to handle fairness. Pnueli and Lichtenstein [Lichtenstein & Pnueli 85] reanalyzed the complexity of checking linear-time formulas and discovered that although the complexity appears exponential in the length of the formula, it is linear in the size of the global state graph. Based on this observation, they argued that the high complexity of linear-time model checking might still be acceptable for short formulas. Emerson and Lei [Emerson & Lei 85] extended their result to show that formulas of the logic CTL^* , which combines both branching-time and linear-time operators, could be checked with essentially the same complexity as formulas of linear temporal logic. Vardi and Wolper have recently [Vardi & Wolper 86] shown how the model checking problem can be formulated in terms of automata, thus relating the model checking approach to the work of Kurshan.

Although the model checking procedure discussed in this paper has already been used to discover some surprising errors in non-trivial programs, more work still remains to be done. Certainly the most serious problem is the *state explosion problem*. In analyzing a system of N processes, the number of states in the global state graph may grow exponentially with N . Recent research indicates, however, that it may be possible to avoid this problem in some important cases. For instance, techniques developed in [Clarke et al 86b] may reduce the size of the state graph that needs to be searched when many of the processes are identical. It may also be possible to exploit the hierarchical structure of a complex concurrent program in order to reduce the number of states that need to be considered at any one level of abstraction [Mishra & Clarke 85].

This survey is organized as follows: Section 2 describes the syntax and semantics of the temporal logics that are used in this paper. In Section 3 we state the *model checking problem* and give an efficient algorithm for checking simple branching-time formulas. In Section 4 we discuss the issue of fairness and show how the algorithm of Section 3 can be extended to include *fairness constraints*. Section 5 demonstrates how the model checking algorithm can be used to debug a simple mutual exclusion program. In Section 6 we describe some alternative approaches for verifying systems of finite state concurrent processes. We analyze the complexity of checking linear temporal logic formulas and outline the techniques of Pnueli and Lichtenstein [Lichtenstein & Pnueli 85] and Vardi and Wolper [Vardi & Wolper 86]. Additional applications to circuit and protocol verification are discussed in Section 7. The paper concludes in Section 8 with a discussion of some of the important remaining research problems like the *state explosion problem*.

2. Computation Tree Logics

In this paper finite state programs are modelled by labelled state-transition graphs, called *Kripke structures* [Hughes & Creswell 77]. If some state is designated as the *initial* state, then the Kripke structure can be unwound into an infinite tree with that state as the root. Since paths in the tree represent possible computations of the program, we will refer to the infinite tree obtained in this manner as the *computation tree* of the program. Temporal logics may differ according to how they handle branching in the underlying computation tree. In linear temporal logic, operators are provided for describing events along a single computation path. In a branching-time logic the temporal operators quantify over the paths that are possible from a given state. The computation tree logic CTL^* ([Emerson & Clarke 81], [Emerson & Halpern 83], [Clarke et al 86a]) combines both branching-time and linear-time operators; a path quantifier, either A ("for all computation paths") or E ("for some computation path") can prefix an assertion composed of arbitrary combinations of the usual linear-time

operators G ("always"), F ("sometimes"), X ("nexttime"), and U ("until"). The remainder of this section gives a precise description of the syntax and semantics of these logics.

There are two types of formulas in CTL^* : *state formulas* (which are true in a specific state) and *path formulas* (which are true along a specific path). Let AP be the set of atomic proposition names. A state formula is either:

- A , if $A \in AP$.
- If f and g are state formulas, then $\neg f$ and $f \vee g$ are state formulas.
- If f is a path formula, then $F(f)$ is a state formula.

A path formula is either:

- A state formula.
- If f and g are path formulas, then $\neg f$, $f \vee g$, Xf , and fUg are path formulas.

CTL^* is the set of state formulas generated by the above rules.

CTL ([Ben-Ari et al 83], [Clarke & Emerson 81]) is a restricted subset of CTL^* that permits only branching-time operators--each path quantifier must be immediately followed by exactly one of the operators G , F , X , or U . More precisely, CTL is the subset of CTL^* that is obtained if the path formulas are restricted as follows:

- If f and g are state formulas, then Xf and fUg are path formulas.
- If f is a path formula, then so is $\neg f$.

Linear temporal logic (LTL), on the other hand, will consist of formulas that have the form Af where f is a path formula in which the only state subformulas that are permitted are atomic propositions. More formally, a path formula is either

- An atomic proposition
- If f and g are path formulas, then $\neg f$, $f \vee g$, Xf , and fUg are path formulas.

We define the semantics of CTL^* with respect to a structure $M = \langle S, R, L \rangle$, where

- S is a set of states.
- $R \subseteq S \times S$ is the transition relation, which must be total. We write $s_1 \rightarrow s_2$ to indicate that $(s_1, s_2) \in R$.
- $L: S \rightarrow \mathcal{P}(AP)$ is a function that labels each state with a set of atomic propositions true in that state.

Unless otherwise stated, all of our results apply only to *finite* Kripke structures.

We define a *path* in M to be a sequence of states, $\pi = s_0, s_1, \dots$ such that for every $i \geq 0$, $s_i \rightarrow s_{i+1}$. π^i will denote the *suffix* of π starting at s_i .

We use the standard notation to indicate that a state formula f holds in a structure: $M, s \models f$ means that f holds at state s in structure M . Similarly, if f is a path formula, $M, \pi \models f$ means that f holds along path π in structure M . The relation \models is defined inductively as follows (assuming that f_1 and f_2 are state formulas and g_1 and g_2 are path formulas):

1. $s \models A \iff A \in I(s).$
2. $s \models \neg f_1 \iff s \not\models f_1.$
3. $s \models f_1 \vee f_2 \iff s \models f_1 \text{ or } s \models f_2.$
4. $s \models E(g_1) \iff \text{there exists a path } \pi \text{ starting with } s \text{ such that } \pi \models g_1.$
5. $\pi \models f_1 \iff s \text{ is the first state of } \pi \text{ and } s \models f_1.$
6. $\pi \models \neg g_1 \iff \pi \not\models g_1.$
7. $\pi \models g_1 \vee g_2 \iff \pi \models g_1 \text{ or } \pi \models g_2.$
8. $\pi \models Xg_1 \iff \pi^1 \models g_1.$
9. $\pi \models g_1 U g_2 \iff \text{there exists a } k \geq 0 \text{ such that } \pi^k \models g_2 \text{ and for all } 0 \leq j < k, \pi^j \models g_1.$

We will also use the following abbreviations in writing CTL* (CTL and LTL) formulas:

- $f \wedge g \equiv \neg(\neg f \vee \neg g)$
- $Ef \equiv \text{true } Uf$
- $A(f) \equiv \neg E(\neg f)$
- $Gf \equiv \neg F\neg f.$

In ([Lamport 80], [Emerson & Halpern 83]) it is shown that the three logics discussed in this section have different expressive powers. For example, there is no CTL formula that is equivalent to the LTL formula $A(FG p)$. Likewise, there is no LTL formula that is equivalent to the CTL formula $AG(EF p)$. The disjunction of these two formulas $A(FG p) \vee AG(EF p)$ is a CTL* formula that is not expressible in either CTL or LTL.

3. The CTL Model Checking Algorithm

Let $M = (S, R, I)$ be a finite Kripke structure. Assume that we want to determine which states in S satisfy the CTL formula f_0 . We will design our algorithm to operate in stages. The first stage processes all subformulas of f_0 of length 1, the second stage processes all subformulas of length 2, and so on. At

the end of the i^{th} stage, each state will be labeled with the set of all subformulas of length less than or equal to i that are true in the state. We let the expression $label(s)$ denote this set for state s . When the algorithm terminates at the end of stage $n = length(f_0)$, we see that for all states and for all subformulas f of f_0 , $M, s \models f$ iff $f \in label(s)$.

Observe that AX can be expressed in terms of EX and that AU can be expressed in terms of EU and EG:

$$AX f_1 \equiv \neg EX \neg f_1$$

$$A[f_1 U f_2] \equiv \neg (E[\neg f_2 U (\neg f_1 \wedge \neg f_2)] \vee EG(\neg f_2)).$$

Thus, for the stage i algorithm it is sufficient to be able to handle six cases, depending on whether f is atomic or has one of the following forms: $\neg f_1$, $f_1 \vee f_2$, $EX f_1$, $E[f_1 U f_2]$ or $EG f_1$.

We will only consider the last two cases, since the others are straightforward.

To handle formulas of the form $f = E[f_1 U f_2]$ we first find all of those states that are labeled with f_2 . We then work backwards using the converse of the transition relation R and find all of those states that can be reached by a path in which each state is labeled with f_1 . All such states should be labeled with f . This step requires time $O(|S| + |R|)$.

The case in which $f = EG f_1$ is slightly more complicated and depends on the following observation.

Lemma 1: Let M' be obtained from M by deleting from S all of those states at which f_1 does not hold and restricting R and L accordingly. Thus, $M' = (S', R', L')$ where $S' = \{s \in S \mid M, s \models f_1\}$, $R' = R|_{S' \times S'}$, and $L' = L|_{S'}$. Then $M, s \models EG f_1$ iff the following two conditions are satisfied:

1. $s \in S'$
2. there exists a path in S' that leads from s to some node t in a *non-trivial* strongly connected component² of the graph (S', R') .

Proof: Assume that $M, s \models EG f_1$. Clearly $s \in S'$. Let π be an infinite path starting at s such that f_1 holds at each state on π . Since M is finite, it must be possible to write π as $\pi = \pi_1 \pi_2$ where π_1 is a finite initial segment and π_2 is an infinite suffix of π with the property that each state on π_2 occurs

² A non-trivial strongly connected component is one that contains at least one cycle.

³ A non-trivial strongly connected component is one that contains at least one cycle.

infinitely often. Obviously π_0 is contained in S' . Let C be the set of states in π_1 . C is a nontrivial strongly connected component of S' . To see this, let s_1 and s_2 be states in C . Pick some instance of s_1 on π_1 . By the way in which π_1 was selected, we know that there is an instance of s_2 further along π_1 . The segment from s_1 to s_2 lies entirely within C and hence within S' . This segment is a finite path from s_1 to s_2 in S' . Thus, both condition (1) and condition (2) are satisfied.

Next, assume that conditions (1) and (2) are satisfied. Let π_1 be the path from s to t . Let π_2 be a finite path of length at least one that leads from t back to t . The existence of π_2 is guaranteed since C is a non-trivial strongly connected component. All of the states on the infinite path $\pi = \pi_1 \pi_2^\omega$ satisfy f_1 . Since π is also a possible path starting at s in M , we see that $M, s \models \text{EG} f_1$. \square

The algorithm for the case of $f = \text{EG} f_1$ follows directly from the lemma. We construct the restricted Kripke structure $M' = (S', R', L')$ as described in the statement of the lemma. We partition the graph (S', R') into strongly connected components and find those states that belong to nontrivial components. We then work backwards using the converse of R and find all of those states that can be reached by a path in which each state is labeled with f_1 . This step also requires time $O(|S| + |R|)$.

In order to handle an arbitrary CTL formula f_0 , we successively apply the state labeling algorithm to the subformulas of f_0 , starting with the shortest, most deeply nested and work outward to include all of f_0 . Since each pass takes time $O(|S| + |R|)$ and since f_0 has $\text{length}(f_0)$ different subformulas, the entire algorithm requires $O(\text{length}(f_0) \cdot (|S| + |R|))$.

Theorem 2: There is an algorithm for determining whether a CTL formula f_0 is true in state s of the structure $M = (S, R, L)$ that runs in time $O(\text{length}(f_0) \cdot (|S| + |R|))$.

4. Fairness Constraints

In verifying concurrent systems, we are occasionally interested only in correctness along *fair* execution sequences. For example, with a system of concurrent processes we may wish to consider only those computation sequences in which each process is executed infinitely often. When dealing with network protocols where processes communicate over an imperfect (or lossy) channel we may also wish to restrict the set of computation sequences; in this case the *unfair* execution sequences are those in which a sender process continuously transmits messages without any reaching the receiver due to erratic behavior by the channel.

Roughly speaking, a fairness condition asserts that requests for service are granted "sufficiently often".

Different concepts of what constitutes a "request" and what "sufficiently often" should mean give rise to a variety of notions of fairness. Indeed, many different types of fairness and approaches to dealing with them have been proposed in the literature; we refer the reader to [Gabbay et al 80], [Lamport 80], [Quielle & Sifakis 82], and [Lehmann et al 81] for more extensive treatments. The text by Francez [Francez 86] also gives an excellent survey of the various types of fairness.

In this section we will show how to extend the CTL model checking algorithm to handle a simple but fundamental type of fairness in which certain predicates must hold infinitely often along every fair path. ([Clarke et al 86a] shows how to handle a richer class of fairness constraints.) In this case it follows from [Emerson & Halpern 83] that correctness of fair executions cannot be expressed in CTL.

In order to handle fairness and still obtain an efficient model checking algorithm we modify the semantics of CTL. The new logic, which we call CTL^F , has the same syntax as CTL. But a structure is now a 4-tuple $M = (S, R, L, F)$ where S, R, L have the same meaning as in the case of CTL, and F is a collection of predicates on S , $F \subseteq 2^S$. A path π is *F-fair* iff the following condition holds: *for each $G \in F$, there are infinitely many states on π which satisfy predicate G .* CTL^F has exactly the same semantics as CTL except that all path quantifiers range over fair paths. The first step in checking CTL^F formulas is to determine the *fair strongly connected components* of the graph of M . A strongly connected component is fair if it contains at least one state from each set in F . Formally, let $F = \{G_1, \dots, G_k\}$ be a collection of subsets of S . A strongly connected component C of the graph of M is *fair* iff for each G_i in F , there is a state $t_i \in (C \cap G_i)$.

Lemma 3: Given any finite structure $M = (S, R, L, F)$ where F is a set of fairness constraints and a state $s_0 \in S$, the following two conditions are equivalent:

1. There exists an *F-fair* path in M starting at s_0 .
2. There exists a fair strongly connected component C of (the graph of) M such that there is a finite path from s_0 to a state $t \in C$.

The proof is straightforward and is given in [Clarke et al 86a]. We next extend our model checking algorithm to CTL^F . We introduce an additional proposition Q , which is true at a state iff there is a fair path starting from that state. This can easily be done, by obtaining the strongly connected components of the graph associated with the structure and marking a component as *fair* if it contains at least one state from each G_i in F . By the above lemma every state in a fair strongly connected component is the start of an infinite fair path. Thus, we label a state with Q iff there is a path from that state to some

node of a fair strongly connected component. As usual we design the algorithm so that after it terminates each state will be labeled with the subformulas of f_0 true in that state. We consider the two interesting cases where f is a subformula of f_0 and either $f = F[f_1 U f_2]$ or $f = FG f_1$. We assume that the states have already been labeled with the immediate subformulas of f by an earlier stage of the algorithm.

1. $f = F[f_1 U f_2]$: f is true in a state iff the CTL formula $F[f_1 U (f_2 \wedge Q)]$ is true in that state, and this can be determined using the CTL model checker. Again, state s is labeled with f iff f is true in that state.
2. $f = FG(f_1)$: To determine if $s \models FG(f_1)$ we use the procedure described in section 3 to check $s \models FG(f_1 \wedge Q)$ in the structure with the additional proposition Q .

It is easy to see that the above algorithm runs in time $O(\text{length}(f_0) \cdot (|S| + |R|))$.

Theorem 4: There is an algorithm for determining whether a CTL^F formula f_0 is true in state s of the structure $M = (S, R, L, F)$ with F as the set of fairness constraints that runs in time $O(\text{length}(f_0) \cdot (|S| + |R|))$.

5. An Example

In this section we illustrate how the model checker can be used to verify a simple, but not entirely trivial, concurrent program. The example is a two process mutual exclusion program that was manually proved correct using linear temporal logic by Owicki and Lamport in [Owicki & Lamport 82]. The program, expressed in a variant of the CSP programming language [Hoare 78], is shown in Figure 5-1. In this version of CSP processes may have global variables (e.g. $p1$ and $p2$), and assignments to such variables are assumed to be atomic. Since our verification technique can only be used to analyze finite state concurrent systems, we require that all variables be boolean and that all messages between processes be signals. Labels (e.g. $NC1$ and $NC2$) are used to indicate that flow of control has reached a particular point in some process. In our example there are two processes $S1$ and $S2$, and each process has three code regions: a *noncritical region* NCi in which the process computes some data values that it wishes to share with the other process, a *trying region* Ti in which the process executes a protocol to obtain entry into the critical section, and a *critical section* CSi in which the process updates shared variables. To prevent a race condition that might result in unpredictable values being assigned to the shared variables, only one process is allowed to be in its critical section at any given time. Note that the two processes are different; hence this is not a *symmetric* solution to the mutual exclusion problem. When the CSP program is compiled a state graph with 77 states is obtained. Although this is not an extremely large state machine, it would nevertheless be quite tedious for a human to debug.

We initially run the verifier without any fairness constraints--See Figure 5-2. We first check to see if both processes are ever in their critical regions at the same time. This property is succinctly expressed by the CTL formula $EF(CS1 \wedge CS2)$. The verifier rapidly determines that the formula is false--hence, the program does guarantee mutual exclusion. Time is measured in 1/60 of a second. The first component measures user cpu time. The second component measures system cpu time. We next check for absence of deadlock. This is expressed by the formula $AG(EF(CS1 \vee CS2))$. The verifier determines that this formula is satisfied: thus, from any state that is reachable from the initial state it is always possible to get to either $CS1$ or $CS2$.

Absence of starvation for process 1 is expressed by the formula $AG(T1 \rightarrow AF CS1)$. This property is not satisfied without a fairness constraint. The reason is quite simple. When we build the global state graph for the program we do not make any assumptions about the relative speeds of the two processes. Thus, the second process can make any number of steps between steps of the first process. In fact, the second process can even run forever, thereby preventing the first process from ever making another step. We can rule out the second type of behavior by means of *fairness constraints* which require that each process be given a chance to execute infinitely often. In Figure 5-3 we restart the verifier with several fairness constraints that prevent either process from **remaining** forever at the same statement while enabled to make a step. Under these assumptions the first process will never starve. However, the possibility of starvation still exists for the second process.

A good solution to the mutual exclusion problem should not require that processes alternate entry into their critical regions: $CS1, CS2, CS1, CS2, \dots$. In order to test that the algorithm given in Figure 5-1 does not require strict alternation, we check the formula

$$AG(CS1 \rightarrow A[CS1 U (\neg CS1 \wedge A[\neg CS1 U CS2]))).$$

This formula asserts that if process 1 enters its critical section and subsequently leaves it, then it cannot enter it again until process 2 has entered its critical section. The verifier determines that the formula is false in less than a second. This example shows how the basic temporal operators, particularly the *until* operator, can be nested to express complicated timing properties.

Finally, the verifier has a counterexample feature (that is not shown in the transcripts). When this feature is enabled and the model checker determines that a formula is false, it will attempt to find a path in the state graph which demonstrates that the negation of the formula is true. For example, if the formula has the form $AG(f)$, our system will produce a path to a state in which $\neg f$ holds. For instance, when the verifier determines that the last formula above is false, it prints out an execution of

```

s :: [
  p1,p2: bool;
  NC1,NC2,T1,T2,T2a,CS1,CS2: label;
  [
    S1,S2: process;
    S1 :: [
      p1 := false;
      *[
        true ->
          <<NC1>> skip; --noncritical section 1
          p1 := true;
          <<T1>>  *[ p2 -> skip];
          <<CS1>> skip; --critical section 1
          p1 := false
        ]
      ]
    ||
    S2 :: [
      p2 := false;
      *[
        true ->
          <<NC2>> skip; --noncritical section2
          p2 := true;
          <<T2>>  *[ p1 ->
            p2 := false;
            <<T2a>>    *[p1 -> skip ];
            p2 := true
          ];
          <<CS2>> skip; --critical section 2
          p2 := false
        ]
      ]
  ]
]

```

Figure 5-1: Two process mutual exclusion program.

CTL MODEL CHECKER (C version 2.5)

$\models \text{EF}(\text{CS1} \ \& \ \text{CS2}).$

The equation is FALSE.

time: (2 4)

$\models \text{AG}(\text{EF}(\text{CS1} \mid \text{CS2})).$

The equation is TRUE.

time: (4 2)

$\models \text{AG}(\text{T1} \rightarrow \text{AF CS1}).$

The equation is FALSE.

time: (17 12)

Figure 5-2: Transcript of model checker execution (without fairness constraint).

Fairness constraint: $\sim \text{NC1}.$

Fairness constraint: $\sim \text{NC2}.$

Fairness constraint: $\sim \text{CS1}.$

Fairness constraint: $\sim \text{CS2}.$

Fairness constraint: $\sim \text{T1} \mid \text{P2}.$

Fairness constraint: $\sim \text{T2} \mid \text{p1}.$

Fairness constraint: $\sim \text{T2} \mid \sim \text{p1} \mid \text{T2a}.$

Fairness constraint: .

$\models \text{AG}(\text{T1} \rightarrow \text{AF CS1}).$

The equation is TRUE.

time: (10 0)

$\models \text{AG}(\text{T2} \rightarrow \text{AF CS2}).$

The equation is FALSE.

time: (29 9)

$\models \text{AG}(\text{CS1} \rightarrow \text{A}[\text{CS1} \cup (\sim \text{CS1} \ \& \ \text{A}[\sim \text{CS1} \cup \text{CS2}]))).$

The equation is FALSE.

time: (38 17)

Figure 5-3: Transcript of model checker execution (with fairness constraint).

the mutual exclusion program in which process 1 enters its critical region, leaves, and reenters without process 2 entering its critical section in the meantime. This feature is quite useful for debugging purposes.

6. Other Approaches

Several papers have considered the model checking problem for linear temporal logic formulas. Let $M = (S, R, L)$ be a Kripke Structure with $s_0 \in S$, and let Af be a linear temporal logic formula. Thus, f is a *restricted path formula* in which the only state subformulas are atomic propositions. We wish to determine if $M, s_0 \models Af$. Notice that $M, s \models Af$ iff $M, s \models \neg E \neg f$. Consequently, it is sufficient to be able to check the truth of formulas of the form Ef where f is a restricted path formula. In general, this problem is PSPACE-complete [Sistla & Clarke 86]. Although the proof of this PSPACE-completeness result is beyond the scope of our survey, it is easy to see that the model checking problem is NP-hard for formulas of the form Ef where f is restricted path formula. We show that the *directed Hamiltonian path problem* is reducible to the problem of determining whether $M, s \models f$ where

- M is a finite structure,
- s is a state in M and
- f is the assertion (using atomic propositions p_1, \dots, p_n):

$$F[Fp_1 \wedge \dots \wedge Fp_n \wedge G(p_1 \rightarrow XG\neg p_1) \wedge \dots \wedge G(p_n \rightarrow XG\neg p_n)].$$

Consider an arbitrary directed graph $G = (V, A)$ where $V = \{v_1, \dots, v_n\}$. We obtain a structure from G by making proposition p_i hold at node v_i and false at all other nodes (for $1 \leq i \leq n$), and by adding a source node u_1 from which all v_i are accessible (but not vice versa) and a sink node u_2 which is accessible from all v_i (but not vice versa). Formally, let the structure $M = (U, B, L)$ consist of

$$U = V \cup \{u_1, u_2\} \text{ where } u_1, u_2 \notin V;$$

$$B = A \cup \{(u_1, v_i) | v_i \in V\} \cup \{(v_i, u_2) | v_i \in V\} \cup \{(u_1, u_2)\}; \text{ and}$$

L is an assignment of propositions to states such that

- p_i is true in v_i for $1 \leq i \leq n$
- p_j is false in v_i for $1 \leq i, j \leq n, i \neq j$
- p_i is false in u_1, u_2 for $1 \leq i \leq n$

It is easy to see that $M, u_1 \models f$ iff there is a directed infinite path in M starting at u_1 which goes through all $v_i \in V$ exactly once and ends in the self loop through u_2 . Note that the formula f in the above construction has essentially the same size as the graph G . Suppose that the length of the formula to be checked was known to be much smaller than the size of the Kripke structure under consideration. Would the complexity still be high in this case? A careful analysis by Lichtenstein and Pnueli [Lichtenstein & Pnueli 85] showed that although the complexity is apparently exponential in the length of the formula, it is linear in the size of the global state graph. We briefly describe their results below.

Let f be a restricted path formula. The *closure* of f , $CL(f)$, is the smallest set of formulas containing f and satisfying:

- $\neg f_1 \in CL(f)$ iff $f_1 \in CL(f)$
- if $f_1 \vee f_2 \in CL(f)$, then $f_1, f_2 \in CL(f)$
- if $Xf_1 \in CL(f)$, then $f_1 \in CL(f)$
- if $\neg Xf_1 \in CL(f)$, then $X\neg f_1 \in CL(f)$
- if $f_1 U f_2 \in CL(f)$, then $f_1, f_2, X[f_1 U f_2] \in CL(f)$

It can be shown that the size of $CL(f)$ is $S \cdot \text{length}(f)$.

An *atom* is a pair $A = (s_A, F_A)$ with $s_A \in S$ and $F_A \subseteq CL(f) \cup AP$ such that:

- for each proposition $Q \in AP$, $Q \in F_A$ iff $Q \in I(s_A)$
- for every $f_1 \in CL(f)$, $f_1 \in F_A$ iff $\neg f_1 \notin F_A$
- for every $f_1, f_2 \in CL(f)$, $f_1 \vee f_2 \in F_A$ iff f_1 or $f_2 \in F_A$
- for every $\neg Xf_1 \in CL(f)$, $\neg Xf_1 \in F_A$ iff $X\neg f_1 \in F_A$
- for every $f_1, f_2 \in CL(f)$, $f_1 U f_2 \in F_A$ iff $f_2 \in F_A$ or $f_1, X[f_1 U f_2] \in F_A$

Now, a graph G is constructed with the set of atoms as the set of vertices. (A, B) is an edge of G iff $(s_A, s_B) \in R$ and for every formula f_1 , if $Xf_1 \in F_A$, then $f_1 \in F_B$. An *eventuality sequence* is an infinite path π in G such that if $f_1 U f_2 \in F_A$ for some atom A on π , then there exists an atom B , reachable from A along π , such that $f_2 \in F_B$.

Lemma 5: $M, s \models Ff$ iff there exists an eventuality sequence starting at an atom (s, F) such that $f \in F$.

A non-trivial strongly connected component C of the graph G is said to be *self-fulfilling* iff for every atom A in C and for every $f_1 \cup f_2 \in F_A$ there exists an atom B in C such that $f_2 \in F_B$.

Lemma 6: $M, s \models E f$ iff there exists an atom $A = (s, F)$ in G such that $f \in F$ and there exists a path in G from A to a self-fulfilling strongly connected component.

Lemma 6 be used as the basis for a linear temporal logic model checking algorithm. This algorithm has the time complexity $O((|S| + |R|) \cdot 2^{\text{length}(f)})$. Lichtenstein and Pnueli further showed how this basic algorithm could be extended to handle a number of different notions of fairness with essentially the same complexity.

An alternative approach due to Vardi and Wolper [Vardi & Wolper 86] exploits the close relationship between linear temporal logic formulas and Büchi automata. A *Büchi automata* is a tuple $A = (\Sigma, S, \rho, S_0, F)$, where

- Σ is an alphabet.
- S is a set of states.
- $\rho : S \times \Sigma \rightarrow 2^S$ is a nondeterministic transition function.
- $S_0 \subseteq S$ is a set of initial states.
- $F \subseteq S$ is a set of designated states.

A *run* of A on an infinite word $w = a_1 a_2 \dots$ is a sequence $s_0 s_1 \dots$ where $s_0 \in S_0$ and $s_i \in \rho(s_{i-1}, a_i)$, for all $i \geq 1$. A run $s_0 s_1 \dots$ is *accepting* if there is some designated state that repeats infinitely often, i.e., for some $s \in F$ there are infinitely many i 's such that $s_i = s$. The infinite word w is *accepted* by A if there is an accepting run of A over w . The set of infinite words accepted by A is denoted $\mathcal{L}(A)$. The following theorem is proved in [Vardi & Wolper 86].

Lemma 7: For every linear temporal formula Δf , a Büchi automata A_f can be constructed, where $\Sigma = 2^{AP}$ and $|S| \leq 2^{\text{length}(f)}$, such that $\mathcal{L}(A_f)$ is exactly the set of computations satisfying the formula f .

A Kripke Structure $M = (S, R, I)$ with initial state $s_0 \in S$ can be viewed as a Büchi automaton $A_M = (\Sigma, S, \{s_0\}, \rho, S)$ where $\Sigma = 2^{AP}$ and $s' \in \rho(s, a)$ iff $(s, s') \in R$ and $a = I(s)$. Note that any infinite run of this automaton is accepting. $\mathcal{L}(A_M)$ is the set of computations of A_M . Thus, in order to determine whether $M, s \models \Delta f$ it is sufficient to check whether $\mathcal{L}(A_M) \cap \mathcal{L}(A_{\neg f})$ is empty. This can be

determined by an automata theoretic construction with essentially the same time complexity as the Pnueli Lichtenstein algorithm.

One of the expected advantages of using linear temporal logic is that fairness constraints can be handled directly. However, if fairness constraints are included as part of the specifications, the formulas that must be checked will in general be quite large. For instance, consider a fairness constraint which requires that progress be made from any state in the program. The formula that expresses this property is

$$\bigwedge_{s \in S} \neg G(at\ s) \rightarrow \langle \text{rest of specification} \rangle,$$

which has size $O(|S|)$. This problem was realized by Lichtenstein and Pnueli and by Vardi and Wolper. They in fact handle fairness by means of fairness constraints in a manner very similar to the way it is handled in [Clarke et al 86a]. Another problem with using linear temporal logic is that in general it is impossible to handle specifications which involve existential path quantifiers. Although it is possible to check simple formulas of the form $E f$ where f is a restricted path formula, it is not possible to check formulas like $AG(EF f)$, which is used to express absence of deadlock in the example in section 5. Moreover, model checking for the full logic CTL^* is no more difficult than for linear temporal logic as was shown by Emerson and Lei [Emerson & Lei 85].

Theorem 8: If we are given an algorithm AL_{LTL} to solve the model checking problem for linear temporal logic, then we can construct an algorithm AL_{CTL^*} for the full logic CTL^* that has the same order of complexity as AL_{LTL} .

7. Applications

Sequential circuit verification is a natural application for the type of verifier discussed in this paper. Bochmann [Bochmann 82] was probably the first to realize the usefulness of temporal logic for describing the behavior of circuits. He verified an implementation of a self-timed arbiter using linear temporal logic and what he called "reachability analysis." The work of Malachi and Owicki [Malachi & Owicki 81] identified additional temporal operators required to express interesting properties of circuits and also gave specifications for a large class of modules used in self-timed circuits. Although these researchers contributed significantly toward developing an adequate notation for expressing the correctness of sequential circuits, the problem of mechanically verifying a circuit remained unsolved.

In [Mishra & Clarke 85] Clarke and Mishra showed how the EMC algorithm could be used to verify various temporal properties of asynchronous circuits. They developed a technique for extracting a state

graph directly from a wire-list description of the circuit (i.e., from a description of the circuit in terms of its components and their interconnections). The model checker was then used to show that state graph satisfied various specifications expressed in temporal logic. In this way they were able to determine that a self-timed queue element described in Seitz' chapter of Mead and Conway [Seitz 80] did not satisfy its specifications. Their work was later extended by Browne, Clarke, Dill, and Mishra [Browne et al 86] who showed, in general, how a mixed gate and switch level circuit simulator could be used to extract a state graph from a structural description of a sequential circuit. The basic simulation algorithm is shown in Figure 7-1. Circuits are usually designed under the assumption that certain input sequences and combinations will not occur. Their program exploits this observation to prevent a combinatorial explosion in the number of states that are generated, by allowing the user to specify a set of conditions under which the inputs can change.

```
{The procedure below uses a hash table that maps node
value assignments to states. To construct the state machine,
call this procedure on a node_value_assignment for the
initial state.}

procedure BuildGraph(Node_value_assignment) return a state
begin
  if there is a state for the node_value_assignment
  already in the table then return the state;
  else
    Create a new state;
    Label state with nodes that have 1 values;
    Store state and node values together in hash table;
    for each possible input assignment do
      Combine current values for internal nodes and input
      assignment into a new node_value_assignment;
      Simulate one step to find a new node assignment;
      Call BuildGraph recursively on new node assignment;
      Add value returned by previous line to successors of
      current state;
    end
  end
end
```

Figure 7-1: Algorithm For Constructing Kripke Structure From Circuit

The circuit simulator in [Browne et al 86] used a unit-delay timing model in which the switching delays of all the transistors and gates are assumed to be equal. While a unit-delay model is satisfactory for synchronous circuits, it may not be appropriate for asynchronous circuits. In [Dill & Clarke 86] Dill and Clarke showed how Kripke structures could be extracted from a gate level description of a circuit under a model of circuit behavior that permitted arbitrary non-zero delays to be associated with the outputs of the gates. The basic idea behind their approach is quite simple. Consider an AND gate with

two inputs, x and y , and a single output z . Assume that the gate is in an unstable configuration with x low, y high, and z high. The Kripke structure for the circuit containing this gate will have a state corresponding to the unstable configuration as shown in Figure 7-2. The state will have a self-loop and a transition to another state representing a stable configuration in which the output is low. Fairness constraints, as described in Section 4, are used to insure that the system doesn't remain in an unstable configuration forever. In the case of the AND gate, it is sufficient to require that infinitely often $z = x \wedge y$.

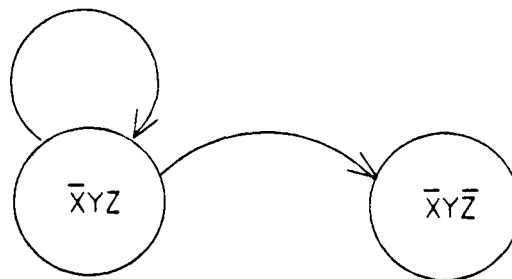


Figure 7-2: Kripke structure for unstable configuration of AND gate.

In practice, the arbitrary delay model is much too conservative. Many circuits are "almost speed independent": They do not appear to be correct under a pure arbitrary delay model, but would work given reasonable assumptions about the relationships between the delays. When the circuit designer has a great deal of control over the magnitudes of circuit delays, exploiting more detailed knowledge of circuit timing can result in smaller and faster circuits. In fact, actual circuits often rely on such assumptions. In [Browne et al 85] and [Dill 86] a method is described for adding such assumptions to a circuit description and incorporating them into the state-graph construction. Possible timing constraints include constant upper and lower bounds on individual delays, and bounds on the differences between delays. Using constraints of this form, one can say for example: "the delay of the first AND gate is between 5 and 10 nanoseconds" or "the delay of the first AND gate is greater than the delay of the second AND gate." The state graph constructed with respect to a particular set of delay assumptions rules out some circuit executions which would be allowed under an arbitrary delay model. Hence, formulas in CTL which might not have been true in an arbitrary delay model may be true with respect to particular delay assumptions (because all the counterexample paths are ruled out by the delay assumptions). This technique was applied to a patented asynchronous queue cell in [Browne et al 85]. The authors determined that the circuit did not meet its specifications under the arbitrary element delay

model. However, under the assumption that the input was slower than two of the circuit gates, they showed that the circuit met its temporal logic specifications.

An alternative approach obtains the state diagram by compilation from a specification of the original (synchronous) circuit in a simple programming language-like notation. Browne and Clarke ([Browne et al 86], [Browne & Clarke 86]) use a Pascal-like state machine description language called SML for this purpose. The language includes the standard control structures *if*, *while*, and *loop/exit*. A *cobegin* statement is also provided for simultaneous execution of statements in lock-step. Since SML programs will ultimately be implemented in hardware, the only data types permitted are *boolean* and (bounded) *integer*. The output of the SML compiler is a deterministic Moore Machine that can be automatically implemented as a PLA, PAL, or a ROM. The output can also be analyzed for correctness using the EMC algorithm. In [Browne 86] Browne describes a specialized version of the EMC algorithm that can check Moore machines much more rapidly than the original algorithm.

Another potential area of application is the verification of network communication protocols. The *alternating bit protocol* [Bartlett et al 69] for reliable transmission of messages by a noisy communication channel is a simple example of such an algorithm. By using the CTL model checking procedure it is possible to determine in a few seconds whether this protocol meets its specifications [Clarke et al 86a]. Sifakis at Grenoble [Quielle & Sifakis 81] and Kurshan at Bell Labs [Kurshan 86] have also considered applications involving network protocols. The delay assumptions mentioned above may be useful for describing the *real-time* behavior of such protocols.

8. Conclusion

Although the verification technique described in this paper has already been used to find some nontrivial errors in circuit designs and communications protocols, more research needs to be done before it will become a truly practical debugging tool for use by system designers. One problem is the expressibility of the underlying temporal logic. For circuit specification *timing diagrams* may be more natural to use than temporal logic formulas. Of course, temporal logic is more general since there is no analogue of negation, disjunction, or conjunction for timing diagrams. It may be possible to either systematically translate timing diagrams into temporal logic formulas or check them directly using an algorithm similar to the one used by the model checker. If so, this would simplify the task of specifying a complicated circuit and also allow the designer to be more confident that specifications actually mean what he thinks they mean.

The most important problem, however, is the *state explosion problem*. There are several different strategies for handling this problem. In verifying asynchronous circuits, for example, buggy circuits sometimes result in much larger state graphs than correct circuits. This happens because the activity in the circuit is much more disordered after an error has occurred. One possible solution in this case is to run the program which builds the state graph and the model checker as co-routines, creating states only as they need to be referenced by the model-checker. In [Dill 86] this technique is called *lazy state generation*, by analogy to lazy evaluation in programming language implementations. By using this method, an error could be discovered and reported after constructing only a small part of the entire state graph; this would not only speed up the verification process, it would also make it possible to verify some circuits which could not be verified if the entire graph had to be constructed.

Another approach to the state explosion problem is to exploit the hierarchical structure of complex finite state concurrent systems. If an appropriate subset of CTL is used ([Mishra & Clarke 85], [Clarke et al 86b]), then lower level subcircuits can be simplified by "hiding" some of their internal nodes (more precisely, making it illegal to use them in temporal logic formulas) and merging groups of states that become indistinguishable into single state. Preliminary research in [Mishra & Clarke 85] indicates that by using this technique it may be possible to cut-down dramatically on the number of states that need to be examined.

Finally, special techniques may be appropriate for concurrent systems that are composed of many identical processes. Consider, for example, a distributed mutual exclusion algorithm for processes arranged in a ring network in which mutual exclusion is guaranteed by means of a token that is passed around the ring ([Dijkstra 85], [Kurshan 85], [Martin 85]). A strategy that is often used for debugging such systems is to consider first a reduced system with one or two processes. If it is possible to show that the reduced system is correct and if the individual processes are really identical, then one is tempted to conclude that the entire system will be correct. In [Clarke et al 86b] an attempt is made to provide a solid theoretical basis that will prevent fallacious conclusions in arguments of this type. The authors describe a temporal logic called *Indexed CTL*^{*}, or *ICTL*^{*} for specifying networks of identical processes. The logic includes all of CTL^{*} with the exception of the nexttime operator; in addition, it permits formulas of the form $\bigwedge_i f(i)$ and $\bigvee_i f(i)$ where $f(i)$ is a formula in which all of the atomic propositions are subscripted by i . A Kripke structure for a family of N identical processes may be obtained as a product of the state graphs of the individual processes. Instances of the same atomic proposition in different processes are distinguished by using the number of the process as a subscript; thus, A_i represents the instance of atomic proposition A associated with process S_i .

Since a closed formula of the new logic cannot contain any atomic propositions with constant index values, it is impossible to refer to a specific process by writing such a formula. Hence, changing the number of processes in a family of identical processes should not effect the truth of a formula in the logic. This intuitive idea is made precise by introducing a new notion of *bisimulation* [Milner 79] between two Kripke structures with the same set of indexed propositions but different sets of index values. It is possible to prove that if two structures correspond in this manner, a closed formula of Indexed CTL^{*} will be true in the initial state of one if and only if it is true in the initial state of the other.

These ideas are illustrated in [Clarke et al 86b] by considering the distributed mutual exclusion algorithm mentioned above. The atomic proposition c_i is true when the i -th process is in its critical region, and the atomic proposition d_i is true when the i -th process is delayed waiting to enter its critical region. A typical requirement for such a system is that a process waiting to enter its critical region will eventually do so. This condition is easily expressed in ICTL^{*} by the formula $\bigwedge_i AG(d_i \Rightarrow AFc_i)$. The results of [Clarke et al 86b] can be used to show that exactly the same ICTL^{*} formulas hold in a network with 1000 processes as hold in a network with two processes. The EMC algorithm can be used to check automatically that the above formula holds in networks of size two and conclude that it will also hold in networks of size 1000. At present this methodology has only been partially automated, however. The bisimulation must be established by hand and this generally requires some representation of the larger Kripke structure. Several researchers are attempting to find a way of automating this phase in a manner that avoids building the larger Kripke structure.

Other techniques for avoiding the state explosion problem are being investigated by Kurshan and Wolper. In Kurshan's system [Kurshan 85] this problem is handled by using a homomorphism to collapse a large state machine into a much smaller one while preserving those properties that are important for verification. Since Kurshan does not use temporal logic formulas for specification, he has no analogue of the indexed formulas or of the bisimulation theorem used in [Clarke et al 86b]. Wolper [Wolper 86] considers a logic somewhat like ICTL^{*} for reasoning about programs that are data-independent; however, his indexed variables range over data elements, not over processes. Also, there is no notion of correspondence between structures in his work. Some ultimate limitations on this type of reasoning are discussed in Apt and Kozen [Apt & Kozen 86].

References

- [Apt & Kozen 86] K. Apt and D. Kozen. Limits for Automatic Verification of Finite-State Concurrent Systems. *Inf. Process. Lett.* 22(6):307-309, 1986.
- [Bartlet et al 69] K.A. Bartlet, R.A. Scantlebury, P.T. Wilkinson. A Note on Reliable Full-Duplex Transmission over Half-Duplex Links. *Communications of the ACM* 12(5):260-261, 1969.
- [Ben-Ari et al 83] M. Ben-Ari, Z. Manna, A. Pnueli. The Temporal Logic of Branching Time. *Acta Informatica* (20):207-226, 1983.
- [Bochmann 82] G. V. Bochmann. Hardware Specification with Temporal Logic: An Example. *IEEE Transactions on Computers* C-31(3), March, 1982.
- [Browne 86] M. C. Browne. An Improved Algorithm for the Automatic Verification of Finite State Systems using Temporal Logic. In *Proceedings of the 1986 Conference on Logic in Computer Science.*, pages 260-267. Cambridge, Massachusetts, June, 1986.
- [Browne & Clarke 86] M. C. Browne, E. M. Clarke. SML: A high level language for the design and verification of Finite State Machines. In *IFIP WG 10.2 International Working Conference from HDL Descriptions to Guaranteed Correct Circuit Designs, Grenoble, France.* IFIP, September, 1986.
- [Browne et al 85] M. C. Browne, E. M. Clarke, D. Dill. Checking the Correctness of Sequential Circuits. In *Proceedings of the 1985 International Conference on Computer Design.* IEEE, Port Chester, New York, October, 1985.

[Browne et al 86] M. Browne, E. Clarke, D. Dill, B. Mishra. Automatic Verification of Sequential Circuits using Temporal Logic. *IEEE Transactions on Computers* C-35(12), December, 1986.

[Browne et al 86b] M. C. Browne, E. M. Clarke, and D. Dill. Automatic Circuit Verification Using Temporal Logic: Two New Examples. G.J. Milne and P.A. Subrahmanyam (editors). *Formal Aspects of VLSI Design*. Elsevier Science Publishers (North Holland), 1986b.

[Clarke & Emerson 81] E.M. Clarke, E.A. Emerson. Synthesis of Synchronization Skeletons for Branching Time Temporal Logic. In *Proc. of the Workshop on Logic of Programs*. Springer-Verlag, Yorktown Heights, NY, 1981.

[Clarke et al 86a] E.M. Clarke, E.A. Emerson, A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems* 8(2):244-263, 1986.

[Clarke et al 86b] E. M. Clarke, O. Grumberg, M. C. Browne. Reasoning about Networks with many identical finite-state processes. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing.*, pages 240-248. ACM, August, 1986.

[Dijkstra 85] E. Dijkstra. Invariance and non-determinacy. In C.A.R. Hoare And J.C. Shepherdson (editors), *Mathematical Logic and Programming Languages*, pages 157-163. Prentice-Hall, 1985.

[Dill 86] D. Dill. A Trace Theoretic Approach to Asynchronous Circuit Verification. Workshop on Design and Implementation of Concurrent programs, Groningen, The Netherlands, November 17-21, 1986

[Dill & Clarke 86] David L. Dill and Edmund M. Clarke. Automatic Verification of Asynchronous Circuits using Temporal Logic. *IEE Proceedings* 133, pt. E(5), September, 1986.

[Emerson & Clarke 81] E.A. Emerson and E.M. Clarke. Characterizing Properties of Parallel Programs as Fixpoints. In *Springer Lecture Notes in Computer Science*. Volume 85: *Proc. of the Seventh International Colloquium on Automata, Languages and Programming*. Springer Verlag, 1981.

[Emerson & Halpern 83] E.A. Emerson, J.Y. Halpern. "'Sometimes" and "Not Never" Revisited: On Branching versus Linear Time". In *Proc. 10th ACM Symp. on Principles of Programming Languages*. 1983.

[Emerson & Lei 85] E.A. Emerson, Chin Laung Lei. Modalities for Model Checking: Branching Time Strikes Back. *Twelfth Symposium on Principles of Programming Languages, New Orleans, La.*, January, 1985.

[Francez 86] N. Francez. *Fairness*. Springer Verlag, 1986.

[Gabbay et al 80] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi, The Temporal Analysis of Fairness. *7th ACM Symposium on Principles of Programming Languages*. :164-173, January, 1980.

[Hoare 78] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM* 21(8), August, 1978.

[Hughes & Creswell 77] G.E. Hughes and M.J. Creswell. *An Introduction to Modal Logic*. Methuen and Co., 1977.

- [Kurshan 85] R.P. Kurshan. Modelling Concurrent Processes. In *Proc. of Symposia in Applied Mathematics*. 1985.
- [Kurshan 86] R.P. Kurshan. *Testing Containment of ω -Regular Languages*. Technical Report 1121-861010-33-TM, Bell Laboratories Technical Memorandum, 1986.
- [Lamport 80] L. Lamport. "Sometimes" is Sometimes "Not Never". In *Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 174-185. Association for Computing Machinery, Las Vegas, January, 1980.
- [Lehmann et al 81] D. Lehmann, A. Pnueli, J. Stavi. Impartiality, Justice, and Fairness: The Ethics of Concurrent Termination. *Automata, Languages, and Programming, Springer Verlag LNCS 115*, 1981.
- [Lichtenstein & Pnueli 85] O. Lichtenstein and A. Pnueli. Checking that Finite State Concurrent Programs Satisfy Their Linear Specification. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*. New Orleans, La., January, 1985.
- [Malachi & Owicki 81] Y. Malachi and S. S. Owicki. Temporal Specifications of Self-Timed Systems. In H.T. Kung, Bob Sproull, and G. Steele (editors), *VLSI Systems and Computations*. 1981.
- [Martin 85] A. Martin. The Design of a Self-Timed Circuit for Distributed Mutual Exclusion. In Henry Fuchs (editor), *Proc. 1985 Chapel Hill Conf. on VLSI*, pages 247-260. 1985.
- [Milner 79] R. Milner. *Lecture Notes in Computer Science*. Volume 92: *A Calculus of Communicating Systems*. Springer-Verlag, 1979.

- [Mishra & Clarke 85] B. Mishra, E.M. Clarke. Hierarchical Verification of Asynchronous Circuits using Temporal Logic. *Theoretical Computer Science* 38:269-291, 1985.
- [Owicki & Lamport 82] S. Owicki, L. Lamport. Proving Liveness Properties of Concurrent Programs. *ACM Transactions on Programming Languages and Systems* 4(3):455-495, July, 1982.
- [Pneuli 77] A. Pneuli. The Temporal Semantics of Concurrent Programs. In *18th Symposium on Foundations of Computer Science*. 1977.
- [Quielle & Sifakis 81] J.P. Quielle, J. Sifakis. "Specification and Verification of Concurrent Systems in CESAR". In *Proc. of the Fifth International Symposium in Programming*. 1981.
- [Quielle & Sifakis 82] J.P. Quielle, J. Sifakis. Fairness and Related Properties in Transition Systems. *IMAG* (292), March, 1982.
- [Seitz 80] C. Seitz. System Timing. *Introduction to VLSI Systems (C. Mead and L. Conway)*. Reading, MA, Addison-Wesley, 1980.
- [Sistla & Clarke 86] A.P. Sistla, E.M. Clarke. Complexity of Propositional Temporal Logics. *Journal of the Association for Computing Machinery* 32(3):733-749, July, 1986.
- [Vardi & Wolper 86] M. Vardi and P. Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *Proceedings of the Conference on Logic in Computer Science*. Boston, Mass., June, 1986.
- [Wolper 86] P. Wolper. Expressing Interesting Properties of Programs in Propositional Temporal Logic. In *Thirteenth ACM Symposium on Principles of Programming Languages*. 1986.

END

FILMED

MARCH, 19 88

DTIC

AD-A188 618

RESEARCH ON AUTOMATIC VERIFICATION OF FINITE-STATE
CONCURRENT SYSTEMS.. (U) CARNEGIE-MELLON UNIV PITTSBURGH
PA DEPT OF COMPUTER SCIENCE.. R E BRYANT ET AL. DEC 87

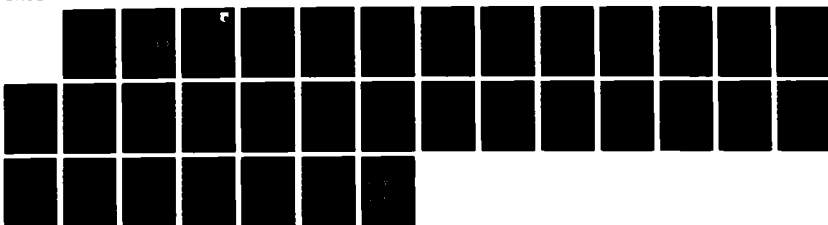
1/1

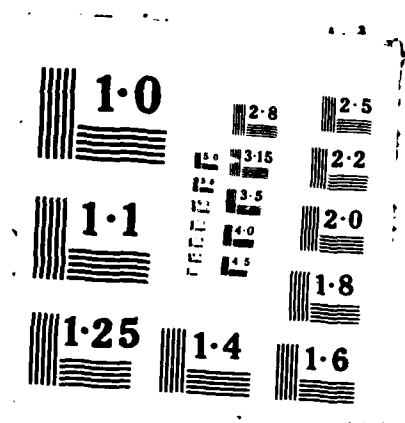
UNCLASSIFIED

CMU-CS-87-105 AFMRL-TR-87-1166

F/G 9/1

NL





AD-A188 618

DTIC ACCESSION NUMBER

LEVEL

PHOTOGRAPH THIS SHEET

INVENTORY

AFWAL-TR-87-1166

DOCUMENT IDENTIFICATION

DEC 1987

This document has been approved
for public release and sale; its
distribution is unlimited.

DISTRIBUTION STATEMENT

ACCESSION FOR

NTIS GRA&I ☒

DTIC TAB ☐

UNANNOUNCED ☐

JUSTIFICATION

BY

DISTRIBUTION /

AVAILABILITY CODES

DIST

AVAIL AND/OR SPECIAL

A-1

DISTRIBUTION STAMP



DTIC
ELECTE
FEB 09 1988
S D
E

DATE ACCESSIONED

DATE RETURNED

88 2 05 102

DATE RECEIVED IN DTIC

REGISTERED OR CERTIFIED NO.

PHOTOGRAPH THIS SHEET AND RETURN TO DTIC-DDAC

AD-A188 618

AFWAL-TR-87-1166

RESEARCH ON AUTOMATIC VERIFICATION OF FINITE-STATE
CONCURRENT SYSTEMS

E.M. Clarke and O. Grumberg

Carnegie-Mellon University
Computer Science Department
Pittsburgh, PA 15213-3890

December 1987

Interim



Approved for Public Release; Distribution is Unlimited

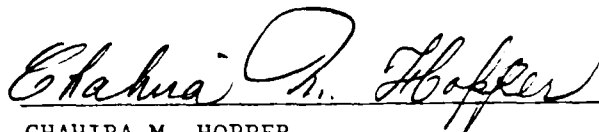
AVIONICS LABORATORY
AIR FORCE WRIGHT AERONAUTICAL LABORATORIES
AIR FORCE SYSTEMS COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OHIO 45433-6543

NOTICE

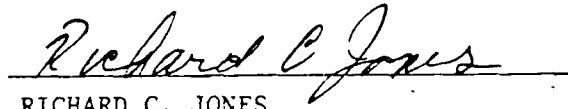
When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely Government-related procurement, the United States Government incurs no responsibility or any obligation whatsoever. The fact that the Government may have formulated or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication, or otherwise in any manner construed, as licensing the holder, or any other person or corporation; or as conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

This report has been reviewed by the Office of Public Affairs (ASD/PA) and is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.

This technical report has been reviewed and is approved for publication.



CHAHIRA M. HOPPER
Project Engineer



RICHARD C. JONES
Ch, Advanced Systems Research Gp
Information Processing Technology Br

FOR THE COMMANDER



EDWARD L. GLIATTI
Ch, Information Processing Technology Br
Systems Avionics Div

If your address has changed, if you wish to be removed from our mailing list, or if the addressee is no longer employed by your organization please notify AFWAL/AAAT, Wright-Patterson AFB, OH 45433-6543 to help us maintain a current mailing list.

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.

SECURITY CLASSIFICATION OF THIS PAGE

Form Approved
OMB No 0704-0188

DD Form 1473, JUN 86

Previous editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

Table of Contents

1. Introduction	1
2. Computation Tree Logics	3
3. The CTL Model Checking Algorithm	5
4. Fairness Constraints	7
5. An Example	9
6. Other Approaches	13
7. Applications	16
8. Conclusion	19

List of Figures

Figure 5-1:	Two process mutual exclusion program.	11
Figure 5-2:	Transcript of model checker execution (without fairness constraint).	12
Figure 5-3:	Transcript of model checker execution (with fairness constraint).	12
Figure 7-1:	Algorithm For Constructing Kripke Structure From Circuit	17
Figure 7-2:	Kripke structure for unstable configuration of AND gate.	18

Research On Automatic Verification Of Finite-State Concurrent Systems

E. M. Clarke and O. Grumberg
Carnegie Mellon University, Pittsburgh

1. Introduction

Temporal logics were first developed by philosophers for reasoning about the ordering of events in time without introducing time explicitly [Hughes & Creswell 77]. Although a number of different temporal logics have been studied, most have an operator like $G(f)$ that is true in the present if f is always true in the future (i.e., if f is globally true). To assert that two events e_1 and e_2 never occur at the same time, one would write $G(\neg e_1 \vee \neg e_2)$. Temporal logics are often classified according to whether time is assumed to have a *linear* or a *branching* structure. This classification may occasionally be misleading since some temporal logics combine both linear-time and branching-time operators. Instead, we will adopt the approach used in [Emerson & Halpern 83] that permits both types of logics to be treated within a single semantical framework. In this paper the meaning of a temporal logic formula will always be determined with respect to a labelled state transition graph; for historical reasons such structures are called *Kripke models* [Hughes & Creswell 77].

Pnueli was apparently the first person to use temporal logic for specifying and verifying concurrent programs [Pnueli 77]. His approach involved proving desired properties of the program under consideration from a set of program axioms that described the behavior of the individual statements in the program. Proofs were usually constructed by hand, and this task was in general quite tedious. Since many concurrent programs can be viewed as communicating finite state machines, there was a strong possibility that at least some of these programs could be automatically verified. The first verification technique to exploit this observation was the *CTL model checking procedure* developed by Clarke and Emerson in [Clarke & Emerson 81]. Their algorithm was polynomial in both the size of the model determined by the program under consideration and in the length of its specification in temporal logic. They also showed how *fairness* [Gabbay et al 80] could be handled without changing the complexity of their algorithm. Handling fairness was an important step since the correctness of many concurrent

This research was partially supported by NSF Grant MCS-82-16706. The second author, O. Grumberg, is currently on leave from Technion, Haifa and is partially supported by a Weinmann postdoctoral fellowship.

algorithms depends critically on some assumption of this type; for example, absence of starvation in a mutual exclusion algorithm may depend on the assumption that each process makes progress infinitely often.

At roughly the same time Quielle and Sifakis [Quielle & Sifakis 81] gave a model checking algorithm for a similar branching-time logic, but they did not analyze its complexity or show how to handle an interesting notion of fairness. Later Clarke, Emerson, and Sistla [Clarke et al 86a] devised an improved algorithm that was linear in the product of the length of the formula and in the size of the global state graph. Sistla and Clarke [Sistla & Clarke 86] analyzed the model checking problem for a variety of other temporal logics and showed, in particular, that for linear temporal logic the problem was PSPACE complete.

A number of papers have shown how the temporal logic model checking procedure can be used for verifying network protocols and sequential circuits ([Clarke et al 86a], [Mishra & Clarke 85], [Browne et al 86], [Dill & Clarke 86], [Browne et al 85], [Browne & Clarke 86], [Browne et al 86b]). In the case of sequential circuits two approaches have been developed for obtaining state transition graphs to analyze. *The first approach extracts a state graph directly from the circuit under an appropriate timing model of circuit behavior.* The second approach obtains a state transition graph by compilation from a high level representation of the circuit in a Pascal-like programming language. In practice the model checking procedure is able to check state transition graphs at a rate of 100 states per second for formulas of reasonable length. It has been used successfully to find previously unknown errors in published designs of asynchronous circuits.

Alternative approaches have been proposed by a number of other researchers. The approach used by Kurshan [Kurshan 86] involves checking *inclusion* between two automata on infinite tapes. The first machine represents the system that is being verified; the second represents its specification. Automata on infinite tapes are used in order to handle fairness. Pnueli and Lichtenstein [Lichtenstein & Pnueli 85] reanalyzed the complexity of checking linear-time formulas and discovered that although the complexity appears exponential in the length of the formula, it is linear in the size of the global state graph. Based on this observation, they argued that the high complexity of linear-time model checking might still be acceptable for short formulas. Emerson and Lei [Emerson & Lei 85] extended their result to show that formulas of the logic CTL^* , which combines both branching-time and linear-time operators, could be checked with essentially the same complexity as formulas of linear temporal logic. Vardi and Wolper have recently [Vardi & Wolper 86] shown how the model checking problem can be formulated in terms of automata, thus relating the model checking approach to the work of Kurshan.

Although the model checking procedure discussed in this paper has already been used to discover some surprising errors in non-trivial programs, more work still remains to be done. Certainly the most serious problem is the *state explosion problem*. In analyzing a system of N processes, the number of states in the global state graph may grow exponentially with N . Recent research indicates, however, that it may be possible to avoid this problem in some important cases. For instance, techniques developed in [Clarke et al 86b] may reduce the size of the state graph that needs to be searched when many of the processes are identical. It may also be possible to exploit the hierarchical structure of a complex concurrent program in order to reduce the number of states that need to be considered at any one level of abstraction [Mishra & Clarke 85].

This survey is organized as follows: Section 2 describes the syntax and semantics of the temporal logics that are used in this paper. In Section 3 we state the *model checking problem* and give an efficient algorithm for checking simple branching-time formulas. In Section 4 we discuss the issue of fairness and show how the algorithm of Section 3 can be extended to include *fairness constraints*. Section 5 demonstrates how the model checking algorithm can be used to debug a simple mutual exclusion program. In Section 6 we describe some alternative approaches for verifying systems of finite state concurrent processes. We analyze the complexity of checking linear temporal logic formulas and outline the techniques of Pnueli and Lichtenstein [Lichtenstein & Pnueli 85] and Vardi and Wolper [Vardi & Wolper 86]. Additional applications to circuit and protocol verification are discussed in Section 7. The paper concludes in Section 8 with a discussion of some of the important remaining research problems like the *state explosion problem*.

2. Computation Tree Logics

In this paper finite state programs are modelled by labelled state-transition graphs, called *Kripke structures* [Hughes & Creswell 77]. If some state is designated as the *initial* state, then the Kripke structure can be unwound into an infinite tree with that state as the root. Since paths in the tree represent possible computations of the program, we will refer to the infinite tree obtained in this manner as the *computation tree* of the program. Temporal logics may differ according to how they handle branching in the underlying computation tree. In linear temporal logic, operators are provided for describing events along a single computation path. In a branching-time logic the temporal operators quantify over the paths that are possible from a given state. The computation tree logic CTL* ([Emerson & Clarke 81], [Emerson & Halpern 83], [Clarke et al 86a]) combines both branching-time and linear-time operators; a path quantifier, either A ("for all computation paths") or E ("for some computation path") can prefix an assertion composed of arbitrary combinations of the usual linear-time

operators G ("always"), F ("sometimes"), X ("nexttime"), and U ("until"). The remainder of this section gives a precise description of the syntax and semantics of these logics.

There are two types of formulas in CTL^* : *state formulas* (which are true in a specific state) and *path formulas* (which are true along a specific path). Let AP be the set of atomic proposition names. A state formula is either:

- A , if $A \in AP$.
- If f and g are state formulas, then $\neg f$ and $f \vee g$ are state formulas.
- If f is a path formula, then $F(f)$ is a state formula.

A path formula is either:

- A state formula.
- If f and g are path formulas, then $\neg f$, $f \vee g$, Xf , and fUg are path formulas.

CTL^* is the set of state formulas generated by the above rules.

CTL ([Ben-Ari et al 83], [Clarke & Emerson 81]) is a restricted subset of CTL^* that permits only branching-time operators--each path quantifier must be immediately followed by exactly one of the operators G , F , X , or U . More precisely, CTL is the subset of CTL^* that is obtained if the path formulas are restricted as follows:

- If f and g are state formulas, then Xf and fUg are path formulas.
- If f is a path formula, then so is $\neg f$.

Linear temporal logic (LTL), on the other hand, will consist of formulas that have the form Af where f is a path formula in which the only state subformulas that are permitted are atomic propositions. More formally, a path formula is either

- An atomic proposition
- If f and g are path formulas, then $\neg f$, $f \vee g$, Xf , and fUg are path formulas.

We define the semantics of CTL^* with respect to a structure $M = \langle S, R, L \rangle$, where

- S is a set of states.
- $R \subseteq S \times S$ is the transition relation, which must be total. We write $s_1 \rightarrow s_2$ to indicate that $(s_1, s_2) \in R$.
- $L: S \rightarrow \mathcal{P}(AP)$ is a function that labels each state with a set of atomic propositions true in that state.

Unless otherwise stated, all of our results apply only to *finite* Kripke structures.

We define a *path* in M to be a sequence of states, $\pi = s_0, s_1, \dots$ such that for every $i \geq 0$, $s_i \rightarrow s_{i+1}$. π^i will denote the *suffix* of π starting at s_i .

We use the standard notation to indicate that a state formula f holds in a structure: $M, s \models f$ means that f holds at state s in structure M . Similarly, if f is a path formula, $M, \pi \models f$ means that f holds along path π in structure M . The relation \models is defined inductively as follows (assuming that f_1 and f_2 are state formulas and g_1 and g_2 are path formulas):

1. $s \models A \iff A \in I(s).$
2. $s \models \neg f_1 \iff s \not\models f_1.$
3. $s \models f_1 \vee f_2 \iff s \models f_1 \text{ or } s \models f_2.$
4. $s \models F(g_1) \iff \text{there exists a path } \pi \text{ starting with } s \text{ such that } \pi \models g_1.$
5. $\pi \models f_1 \iff s \text{ is the first state of } \pi \text{ and } s \models f_1.$
6. $\pi \models \neg g_1 \iff \pi \not\models g_1.$
7. $\pi \models g_1 \vee g_2 \iff \pi \models g_1 \text{ or } \pi \models g_2.$
8. $\pi \models Xg_1 \iff \pi^1 \models g_1.$
9. $\pi \models g_1 U g_2 \iff \text{there exists a } k \geq 0 \text{ such that } \pi^k \models g_2 \text{ and for all } 0 \leq j < k, \pi^j \models g_1.$

We will also use the following abbreviations in writing CTL* (CTL and LTL) formulas:

- $f \wedge g \equiv \neg(\neg f \vee \neg g)$
- $Ff \equiv \text{true } U f$
- $A(f) \equiv \neg E(\neg f)$
- $Gf \equiv \neg F\neg f.$

In ([Lamport 80], [Emerson & Halpern 83]) it is shown that the three logics discussed in this section have different expressive powers. For example, there is no CTL formula that is equivalent to the LTL formula $A(FG p)$. Likewise, there is no LTL formula that is equivalent to the CTL formula $AG(EF p)$. The disjunction of these two formulas $A(FG p) \vee AG(EF p)$ is a CTL* formula that is not expressible in either CTL or LTL.

3. The CTL Model Checking Algorithm

Let $M = (S, R, I)$ be a finite Kripke structure. Assume that we want to determine which states in S satisfy the CTL formula f_0 . We will design our algorithm to operate in stages: The first stage processes all subformulas of f_0 of length 1, the second stage processes all subformulas of length 2, and so on. At

the end of the i^{th} stage, each state will be labeled with the set of all subformulas of length less than or equal to i that are true in the state. We let the expression $label(s)$ denote this set for state s . When the algorithm terminates at the end of stage $n = length(f_0)$, we see that for all states and for all subformulas f of f_0 , $M, s \models f$ iff $f \in label(s)$.

Observe that AX can be expressed in terms of EX and that AU can be expressed in terms of EU and EG:

$$AX f_1 \equiv \neg EX \neg f_1$$

$$A[f_1 U f_2] \equiv \neg (E[\neg f_2 U (\neg f_1 \wedge \neg f_2)] \vee EG(\neg f_2)).$$

Thus, for the stage i algorithm it is sufficient to be able to handle six cases, depending on whether f is atomic or has one of the following forms: $\neg f_1$, $f_1 \vee f_2$, $EX f_1$, $E[f_1 U f_2]$ or $EG f_1$.

We will only consider the last two cases, since the others are straightforward.

To handle formulas of the form $f = E[f_1 U f_2]$ we first find all of those states that are labeled with f_2 . We then work backwards using the converse of the transition relation R and find all of those states that can be reached by a path in which each state is labeled with f_1 . All such states should be labeled with f . This step requires time $O(|S| + |R|)$.

The case in which $f = EG f_1$ is slightly more complicated and depends on the following observation.

Lemma 1: Let M' be obtained from M by deleting from S all of those states at which f_1 does not hold and restricting R and L accordingly. Thus, $M' = (S', R', L')$ where $S' = \{s \in S \mid M, s \models f_1\}$, $R' = R|_{S' \times S'}$, and $L' = L|_{S'}$. Then $M, s \models EG f_1$ iff the following two conditions are satisfied:

1. $s \in S'$
2. there exists a path in S' that leads from s to some node t in a *non-trivial*² strongly connected component² of the graph (S', R') .

Proof: Assume that $M, s \models EG f_1$. Clearly $s \in S'$. Let π be an infinite path starting at s such that f_1 holds at each state on π . Since M is finite, it must be possible to write π as $\pi = \pi_1 \pi_2$ where π_1 is a finite initial segment and π_2 is an infinite suffix of π with the property that each state on π_2 occurs

² A non-trivial strongly connected component is one that contains at least one cycle.

infinitely often. Obviously π_0 is contained in S' . Let C be the set of states in π_1 . C is a nontrivial strongly connected component of S' . To see this, let s_1 and s_2 be states in C . Pick some instance of s_1 on π_1 . By the way in which π_1 was selected, we know that there is an instance of s_2 further along π_1 . The segment from s_1 to s_2 lies entirely within C and hence within S' . This segment is a finite path from s_1 to s_2 in S' . Thus, both condition (1) and condition (2) are satisfied.

Next, assume that conditions (1) and (2) are satisfied. Let π_1 be the path from s to t . Let π_2 be a finite path of length at least one that leads from t back to t . The existence of π_2 is guaranteed since C is a non-trivial strongly connected component. All of the states on the infinite path $\pi = \pi_1 \pi_2^\omega$ satisfy f_1 . Since π is also a possible path starting at s in M , we see that $M, s \models \text{EG} f_1$. \square

The algorithm for the case of $f = \text{EG} f_1$ follows directly from the lemma. We construct the restricted Kripke structure $M' = (S', R', L')$ as described in the statement of the lemma. We partition the graph (S', R') into strongly connected components and find those states that belong to nontrivial components. We then work backwards using the converse of R and find all of those states that can be reached by a path in which each state is labeled with f_1 . This step also requires time $O(|S| + |R|)$.

In order to handle an arbitrary CTL formula f_0 , we successively apply the state labeling algorithm to the subformulas of f_0 , starting with the shortest, most deeply nested and work outward to include all of f_0 . Since each pass takes time $O(|S| + |R|)$ and since f_0 has $\text{length}(f_0)$ different subformulas, the entire algorithm requires $O(\text{length}(f_0) \cdot (|S| + |R|))$.

Theorem 2: There is an algorithm for determining whether a CTL formula f_0 is true in state s of the structure $M = (S, R, L)$ that runs in time $O(\text{length}(f_0) \cdot (|S| + |R|))$.

4. Fairness Constraints

In verifying concurrent systems, we are occasionally interested only in correctness along *fair* execution sequences. For example, with a system of concurrent processes we may wish to consider only those computation sequences in which each process is executed infinitely often. When dealing with network protocols where processes communicate over an imperfect (or lossy) channel we may also wish to restrict the set of computation sequences; in this case the *unfair* execution sequences are those in which a sender process continuously transmits messages without any reaching the receiver due to erratic behavior by the channel.

Roughly speaking, a fairness condition asserts that requests for service are granted "sufficiently often".

Different concepts of what constitutes a "request" and what "sufficiently often" should mean give rise to a variety of notions of fairness. Indeed, many different types of fairness and approaches to dealing with them have been proposed in the literature; we refer the reader to [Gabbay et al 80], [Lamport 80], [Quielle & Sifakis 82], and [Lehmann et al 81] for more extensive treatments. The text by Francez [Francez 86] also gives an excellent survey of the various types of fairness.

In this section we will show how to extend the CTL model checking algorithm to handle a simple but fundamental type of fairness in which certain predicates must hold infinitely often along every fair path. ([Clarke et al 86a] shows how to handle a richer class of fairness constraints.) In this case it follows from [Emerson & Halpern 83] that correctness of fair executions cannot be expressed in CTL.

In order to handle fairness and still obtain an efficient model checking algorithm we modify the semantics of CTL. The new logic, which we call CTL^F , has the same syntax as CTL. But a structure is now a 4-tuple $M = (S, R, L, F)$ where S, R, L have the same meaning as in the case of CTL, and F is a collection of predicates on S , $F \subseteq 2^S$. A path π is *F-fair* iff the following condition holds: *for each $G \in F$, there are infinitely many states on π which satisfy predicate G .* CTL^F has exactly the same semantics as CTL except that all path quantifiers range over fair paths. The first step in checking CTL^F formulas is to determine the *fair strongly connected components* of the graph of M . A strongly connected component is fair if it contains at least one state from each set in F . Formally, let $F = \{G_1, \dots, G_k\}$ be a collection of subsets of S . A strongly connected component C of the graph of M is *fair* iff for each G_i in F , there is a state $t_i \in (C \cap G_i)$.

Lemma 3: Given any finite structure $M = (S, R, L, F)$ where F is a set of fairness constraints and a state $s_0 \in S$, the following two conditions are equivalent:

1. There exists an *F-fair* path in M starting at s_0 .
2. There exists a fair strongly connected component C of (the graph of) M such that there is a finite path from s_0 to a state $t \in C$.

The proof is straightforward and is given in [Clarke et al 86a]. We next extend our model checking algorithm to CTL^F . We introduce an additional proposition Q , which is true at a state iff there is a fair path starting from that state. This can easily be done, by obtaining the strongly connected components of the graph associated with the structure and marking a component as *fair* if it contains at least one state from each G_i in F . By the above lemma every state in a fair strongly connected component is the start of an infinite fair path. Thus, we label a state with Q iff there is a path from that state to some

node of a fair strongly connected component. As usual we design the algorithm so that after it terminates each state will be labeled with the subformulas of f_0 true in that state. We consider the two interesting cases where f is a subformula of f_0 and either $f = F[f_1 U f_2]$ or $f = FG f_1$. We assume that the states have already been labeled with the immediate subformulas of f by an earlier stage of the algorithm.

1. $f = F[f_1 U f_2]$: f is true in a state iff the CTL formula $F[f_1 U (f_2 \wedge Q)]$ is true in that state, and this can be determined using the CTL model checker. Again, state s is labeled with f iff f is true in that state.
2. $f = FG(f_1)$: To determine if $s \models FG(f_1)$ we use the procedure described in section 3 to check $s \models FG(f_1 \wedge Q)$ in the structure with the additional proposition Q .

It is easy to see that the above algorithm runs in time $O(\text{length}(f_0) \cdot (|S| + |R|))$.

Theorem 4: There is an algorithm for determining whether a CTL^F formula f_0 is true in state s of the structure $M = (S, R, L, F)$ with F as the set of fairness constraints that runs in time $O(\text{length}(f_0) \cdot (|S| + |R|))$.

5. An Example

In this section we illustrate how the model checker can be used to verify a simple, but not entirely trivial, concurrent program. The example is a two process mutual exclusion program that was manually proved correct using linear temporal logic by Owicki and Lamport in [Owicki & Lamport 82]. The program, expressed in a variant of the CSP programming language [Hoare 78], is shown in Figure 5-1. In this version of CSP processes may have global variables (e.g. $p1$ and $p2$), and assignments to such variables are assumed to be atomic. Since our verification technique can only be used to analyze finite state concurrent systems, we require that all variables be boolean and that all messages between processes be signals. Labels (e.g. $NC1$ and $NC2$) are used to indicate that flow of control has reached a particular point in some process. In our example there are two processes $S1$ and $S2$, and each process has three code regions: a *noncritical region* NCi in which the process computes some data values that it wishes to share with the other process, a *trying region* Ti in which the process executes a protocol to obtain entry into the critical section, and a *critical section* CSi in which the process updates shared variables. To prevent a race condition that might result in unpredictable values being assigned to the shared variables, only one process is allowed to be in its critical section at any given time. Note that the two processes are different; hence this is not a *symmetric* solution to the mutual exclusion problem. When the CSP program is compiled a state graph with 77 states is obtained. Although this is not an extremely large state machine, it would nevertheless be quite tedious for a human to debug.

We initially run the verifier without any fairness constraints--See Figure 5-2. We first check to see if both processes are ever in their critical regions at the same time. This property is succinctly expressed by the CTL formula $EF(CS1 \wedge CS2)$. The verifier rapidly determines that the formula is false--hence, the program does guarantee mutual exclusion. Time is measured in 1/60 of a second. The first component measures user cpu time. The second component measures system cpu time. We next check for absence of deadlock. This is expressed by the formula $AG(EF(CS1 \vee CS2))$. The verifier determines that this formula is satisfied: thus, from any state that is reachable from the initial state it is always possible to get to either $CS1$ or $CS2$.

Absence of starvation for process 1 is expressed by the formula $AG(T1 \rightarrow AF CS1)$. This property is not satisfied without a fairness constraint. The reason is quite simple. When we build the global state graph for the program we do not make any assumptions about the relative speeds of the two processes. Thus, the second process can make any number of steps between steps of the first process. In fact, the second process can even run forever, thereby preventing the first process from ever making another step. We can rule out the second type of behavior by means of *fairness constraints* which require that each process be given a chance to execute infinitely often. In Figure 5-3 we restart the verifier with several fairness constraints that prevent either process from **remaining** forever at the same statement while enabled to make a step. Under these assumptions the first process will never starve. However, the possibility of starvation still exists for the second process.

A good solution to the mutual exclusion problem should not require that processes alternate entry into their critical regions: $CS1, CS2, CS1, CS2, \dots$. In order to test that the algorithm given in Figure 5-1 does not require strict alternation, we check the formula

$$AG(CS1 \rightarrow A[CS1 U (\neg CS1 \wedge A[\neg CS1 U CS2]))).$$

This formula asserts that if process 1 enters its critical section and subsequently leaves it, then it cannot enter it again until process 2 has entered its critical section. The verifier determines that the formula is false in less than a second. This example shows how the basic temporal operators, particularly the *until* operator, can be nested to express complicated timing properties.

Finally, the verifier has a counterexample feature (that is not shown in the transcripts). When this feature is enabled and the model checker determines that a formula is false, it will attempt to find a path in the state graph which demonstrates that the negation of the formula is true. For example, if the formula has the form $AG(f)$, our system will produce a path to a state in which $\neg f$ holds. For instance, when the verifier determines that the last formula above is false, it prints out an execution of

```

s :: [
  p1,p2: bool;
  NC1,NC2,T1,T2,T2a,CS1,CS2: label;
  [
    S1,S2: process;
    S1 :: [
      p1 := false;
      *[
        true ->
          <<NC1>> skip; --noncritical section 1
          p1 := true;
          <<T1>>  *[ p2 -> skip];
          <<CS1>> skip; --critical section 1
          p1 := false
        ]
      ]
    ||
    S2 :: [
      p2 := false;
      *[
        true ->
          <<NC2>> skip; --noncritical section2
          p2 := true;
          <<T2>>  *[ p1 ->
            p2 := false;
          <<T2a>>    *[p1 -> skip ];
            p2 := true
          ];
          <<CS2>> skip; --critical section 2
          p2 := false
        ]
      ]
  ]
]

```

Figure 5-1: Two process mutual exclusion program.

CTL MODEL CHECKER (C version 2.5)

$\models \text{EF}(\text{CS1} \ \& \ \text{CS2}).$

The equation is FALSE.

time: (2 4)

$\models \text{AG}(\text{EF}(\text{CS1} \mid \text{CS2})).$

The equation is TRUE.

time: (4 2)

$\models \text{AG}(\text{T1} \rightarrow \text{AF CS1}).$

The equation is FALSE.

time: (17 12)

Figure 5-2: Transcript of model checker execution (without fairness constraint).

Fairness constraint: $\sim \text{NC1}.$

Fairness constraint: $\sim \text{NC2}.$

Fairness constraint: $\sim \text{CS1}.$

Fairness constraint: $\sim \text{CS2}.$

Fairness constraint: $\sim \text{T1} \mid \text{P2}.$

Fairness constraint: $\sim \text{T2} \mid \text{p1}.$

Fairness constraint: $\sim \text{T2} \mid \sim \text{p1} \mid \text{T2a}.$

Fairness constraint: .

$\models \text{AG}(\text{T1} \rightarrow \text{AF CS1}).$

The equation is TRUE.

time: (10 0)

$\models \text{AG}(\text{T2} \rightarrow \text{AF CS2}).$

The equation is FALSE.

time: (29 9)

$\models \text{AG}(\text{CS1} \rightarrow \text{A}[\text{CS1} \cup (\sim \text{CS1} \ \& \ \text{A}[\sim \text{CS1} \cup \text{CS2}]))).$

The equation is FALSE.

time: (38 17)

Figure 5-3: Transcript of model checker execution (with fairness constraint).

the mutual exclusion program in which process 1 enters its critical region, leaves, and reenters without process 2 entering its critical section in the meantime. This feature is quite useful for debugging purposes.

6. Other Approaches

Several papers have considered the model checking problem for linear temporal logic formulas. Let $M = (S, R, I)$ be a Kripke Structure with $s_0 \in S$, and let Af be a linear temporal logic formula. Thus, f is a *restricted path formula* in which the only state subformulas are atomic propositions. We wish to determine if $M, s_0 \models Af$. Notice that $M, s \models Af$ iff $M, s \models \neg E \neg f$. Consequently, it is sufficient to be able to check the truth of formulas of the form Ef where f is a restricted path formula. In general, this problem is PSPACE-complete [Sistla & Clarke 86]. Although the proof of this PSPACE-completeness result is beyond the scope of our survey, it is easy to see that the model checking problem is NP-hard for formulas of the form Ef where f is restricted path formula. We show that the *directed Hamiltonian path problem* is reducible to the problem of determining whether $M, s \models f$ where

- M is a finite structure,
- s is a state in M and
- f is the assertion (using atomic propositions p_1, \dots, p_n):

$$F[Fp_1 \wedge \dots \wedge Fp_n \wedge G(p_1 \rightarrow XG\neg p_1) \wedge \dots \wedge G(p_n \rightarrow XG\neg p_n)].$$

Consider an arbitrary directed graph $G = (V, A)$ where $V = \{v_1, \dots, v_n\}$. We obtain a structure from G by making proposition p_i hold at node v_i and false at all other nodes (for $1 \leq i \leq n$), and by adding a source node u_1 from which all v_i are accessible (but not vice versa) and a sink node u_2 which is accessible from all v_i (but not vice versa). Formally, let the structure $M = (U, B, I)$ consist of

$$U = V \cup \{u_1, u_2\} \text{ where } u_1, u_2 \notin V;$$

$$B = A \cup \{(u_1, v_i) | v_i \in V\} \cup \{(v_i, u_2) | v_i \in V\} \cup \{(u_2, u_2)\}; \text{ and}$$

I is an assignment of propositions to states such that

- p_i is true in v_i for $1 \leq i \leq n$
- p_j is false in v_i for $1 \leq i, j \leq n, i \neq j$
- p_i is false in u_1, u_2 for $1 \leq i \leq n$

It is easy to see that $M, u_1 \models f$ iff there is a directed infinite path in M starting at u_1 which goes through all $v_i \in V$ exactly once and ends in the self loop through u_2 . Note that the formula f in the above construction has essentially the same size as the graph G . Suppose that the length of the formula to be checked was known to be much smaller than the size of the Kripke structure under consideration. Would the complexity still be high in this case? A careful analysis by Lichtenstein and Pnueli [Lichtenstein & Pnueli 85] showed that although the complexity is apparently exponential in the length of the formula, it is linear in the size of the global state graph. We briefly describe their results below.

Let f be a restricted path formula. The *closure* of f , $CL(f)$, is the smallest set of formulas containing f and satisfying:

- $\neg f_1 \in CL(f)$ iff $f_1 \in CL(f)$
- if $f_1 \vee f_2 \in CL(f)$, then $f_1, f_2 \in CL(f)$
- if $Xf_1 \in CL(f)$, then $f_1 \in CL(f)$
- if $\neg Xf_1 \in CL(f)$, then $X\neg f_1 \in CL(f)$
- if $f_1 U f_2 \in CL(f)$, then $f_1, f_2, X[f_1 U f_2] \in CL(f)$

It can be shown that the size of $CL(f)$ is $5 \cdot \text{length}(f)$.

An *atom* is a pair $A = (s_A, F_A)$ with $s_A \in S$ and $F_A \subseteq CL(f) \cup AP$ such that:

- for each proposition $Q \in AP$, $Q \in F_A$ iff $Q \in I(s_A)$
- for every $f_1 \in CL(f)$, $f_1 \in F_A$ iff $\neg f_1 \notin F_A$
- for every $f_1, f_2 \in CL(f)$, $f_1 \vee f_2 \in F_A$ iff f_1 or $f_2 \in F_A$
- for every $\neg Xf_1 \in CL(f)$, $\neg Xf_1 \in F_A$ iff $X\neg f_1 \in F_A$
- for every $f_1, f_2 \in CL(f)$, $f_1 U f_2 \in F_A$ iff $f_2 \in F_A$ or $f_1, X[f_1 U f_2] \in F_A$

Now, a graph G is constructed with the set of atoms as the set of vertices. (A, B) is an edge of G iff $(s_A, s_B) \in R$ and for every formula f_i , if $Xf_i \in F_A$, then $f_i \in F_B$. An *eventuality sequence* is an infinite path π in G such that if $f_1 U f_2 \in F_A$ for some atom A on π , then there exists an atom B , reachable from A along π , such that $f_2 \in F_B$.

Lemma 5: $M, s \models Ff$ iff there exists an eventuality sequence starting at an atom (s, F) such that $f \in F$.

A non-trivial strongly connected component C of the graph G is said to be *self-fulfilling* iff for every atom A in C and for every $f_1 U f_2 \in F_A$ there exists an atom B in C such that $f_2 \in F_B$.

Lemma 6: $M, s \models E f$ iff there exists an atom $A = (s, F)$ in G such that $f \in F$ and there exists a path in G from A to a self-fulfilling strongly connected component.

Lemma 6 be used as the basis for a linear temporal logic model checking algorithm. This algorithm has the time complexity $O((|S| + |R|) \cdot 2^{\text{length}(f)})$. Lichtenstein and Pnueli further showed how this basic algorithm could be extended to handle a number of different notions of fairness with essentially the same complexity.

An alternative approach due to Vardi and Wolper [Vardi & Wolper 86] exploits the close relationship between linear temporal logic formulas and Büchi automata. A *Büchi automaton* is a tuple $A = (\Sigma, S, \rho, S_0, F)$, where

- Σ is an alphabet.
- S is a set of states.
- $\rho : S \times \Sigma \rightarrow 2^S$ is a nondeterministic transition function.
- $S_0 \subseteq S$ is a set of initial states.
- $F \subseteq S$ is a set of designated states.

A *run* of A on an infinite word $w = a_1 a_2 \dots$ is a sequence $s_0 s_1 \dots$ where $s_0 \in S_0$ and $s_i \in \rho(s_{i-1}, a_i)$, for all $i \geq 1$. A run $s_0 s_1 \dots$ is *accepting* if there is some designated state that repeats infinitely often, i.e., for some $s \in F$ there are infinitely many i 's such that $s_i = s$. The infinite word w is *accepted* by A if there is an accepting run of A over w . The set of infinite words accepted by A is denoted $\mathcal{L}(A)$. The following theorem is proved in [Vardi & Wolper 86].

Lemma 7: For every linear temporal formula Δf , a Büchi automaton A_f can be constructed, where $\Sigma = 2^{AP}$ and $|S| \leq 2^{\text{length}(f)}$, such that $\mathcal{L}(A_f)$ is exactly the set of computations satisfying the formula f .

A Kripke Structure $M = (S, R, I)$ with initial state $s_0 \in S$ can be viewed as a Büchi automaton $A_M = (\Sigma, S, \{\rho\}, S)$ where $\Sigma = 2^{AP}$ and $s' \in \rho(s, a)$ iff $(s, s') \in R$ and $a = I(s)$. Note that any infinite run of this automaton is accepting. $\mathcal{L}(A_M)$ is the set of computations of A_M . Thus, in order to determine whether $M, s \models \Delta f$ it is sufficient to check whether $\mathcal{L}(A_M) \cap \mathcal{L}(A_{\neg f})$ is empty. This can be

determined by an automata theoretic construction with essentially the same time complexity as the Pnueli Lichtenstein algorithm.

One of the expected advantages of using linear temporal logic is that fairness constraints can be handled directly. However, if fairness constraints are included as part of the specifications, the formulas that must be checked will in general be quite large. For instance, consider a fairness constraint which requires that progress be made from any state in the program. The formula that expresses this property is

$$\bigwedge_{s \in S} \neg G(at\ s) \rightarrow \langle \text{rest of specification} \rangle,$$

which has size $O(|S|)$. This problem was realized by Lichtenstein and Pnueli and by Vardi and Wolper. They in fact handle fairness by means of fairness constraints in a manner very similar to the way it is handled in [Clarke et al 86a]. Another problem with using linear temporal logic is that in general it is impossible to handle specifications which involve existential path quantifiers. Although it is possible to check simple formulas of the form $E f$ where f is a restricted path formula, it is not possible to check formulas like $AG(EF f)$, which is used to express absence of deadlock in the example in section 5. Moreover, model checking for the full logic CTL^* is no more difficult than for linear temporal logic as was shown by Emerson and Lei [Emerson & Lei 85].

Theorem 8: If we are given an algorithm AL_{LTL} to solve the model checking problem for linear temporal logic, then we can construct an algorithm AL_{CTL^*} for the full logic CTL^* that has the same order of complexity as AL_{LTL} .

7. Applications

Sequential circuit verification is a natural application for the type of verifier discussed in this paper. Bochmann [Bochmann 82] was probably the first to realize the usefulness of temporal logic for describing the behavior of circuits. He verified an implementation of a self-timed arbiter using linear temporal logic and what he called "reachability analysis." The work of Malachi and Owicki [Malachi & Owicki 81] identified additional temporal operators required to express interesting properties of circuits and also gave specifications for a large class of modules used in self-timed circuits. Although these researchers contributed significantly toward developing an adequate notation for expressing the correctness of sequential circuits, the problem of mechanically verifying a circuit remained unsolved.

In [Mishra & Clarke 85] Clarke and Mishra showed how the EMC algorithm could be used to verify various temporal properties of asynchronous circuits. They developed a technique for extracting a state

graph directly from a wire-list description of the circuit (i.e., from a description of the circuit in terms of its components and their interconnections). The model checker was then used to show that state graph satisfied various specifications expressed in temporal logic. In this way they were able to determine that a self-timed queue element described in Seitz' chapter of Mead and Conway [Seitz 80] did not satisfy its specifications. Their work was later extended by Browne, Clarke, Dill, and Mishra [Browne et al 86] who showed, in general, how a mixed gate and switch level circuit simulator could be used to extract a state graph from a structural description of a sequential circuit. The basic simulation algorithm is shown in Figure 7-1. Circuits are usually designed under the assumption that certain input sequences and combinations will not occur. Their program exploits this observation to prevent a combinatorial explosion in the number of states that are generated, by allowing the user to specify a set of conditions under which the inputs can change.

```
{The procedure below uses a hash table that maps node
value assignments to states. To construct the state machine,
call this procedure on a node_value_assignment for the
initial state.}

procedure BuildGraph(Node_value_assignment) return a state
begin
  if there is a state for the node_value_assignment
  already in the table then return the state;
  else
    Create a new state;
    Label state with nodes that have 1 values;
    Store state and node values together in hash table;
    for each possible input assignment do
      Combine current values for internal nodes and input
      assignment into a new node_value_assignment;
      Simulate one step to find a new node assignment;
      Call BuildGraph recursively on new node assignment;
      Add value returned by previous line to successors of
      current state;
    end
  end
end
```

Figure 7-1: Algorithm For Constructing Kripke Structure From Circuit

The circuit simulator in [Browne et al 86] used a unit-delay timing model in which the switching delays of all the transistors and gates are assumed to be equal. While a unit-delay model is satisfactory for synchronous circuits, it may not be appropriate for asynchronous circuits. In [Dill & Clarke 86] Dill and Clarke showed how Kripke structures could be extracted from a gate level description of a circuit under a model of circuit behavior that permitted arbitrary non-zero delays to be associated with the outputs of the gates. The basic idea behind their approach is quite simple. Consider an AND gate with

two inputs, x and y , and a single output z . Assume that the gate is in an unstable configuration with x low, y high, and z high. The Kripke structure for the circuit containing this gate will have a state corresponding to the unstable configuration as shown in Figure 7-2. The state will have a self-loop and a transition to another state representing a stable configuration in which the output is low. Fairness constraints, as described in Section 4, are used to insure that the system doesn't remain in an unstable configuration forever. In the case of the AND gate, it is sufficient to require that infinitely often $z = x \wedge y$.

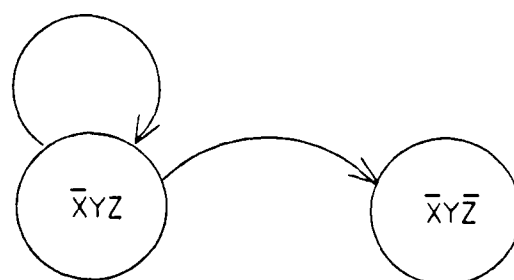


Figure 7-2: Kripke structure for unstable configuration of AND gate.

In practice, the arbitrary delay model is much too conservative. Many circuits are "almost speed independent": They do not appear to be correct under a pure arbitrary delay model, but would work given reasonable assumptions about the relationships between the delays. When the circuit designer has a great deal of control over the magnitudes of circuit delays, exploiting more detailed knowledge of circuit timing can result in smaller and faster circuits. In fact, actual circuits often rely on such assumptions. In [Browne et al 85] and [Dill 86] a method is described for adding such assumptions to a circuit description and incorporating them into the state-graph construction. Possible timing constraints include constant upper and lower bounds on individual delays, and bounds on the **differences** between delays. Using constraints of this form, one can say for example: "the delay of the first AND gate is between 5 and 10 nanoseconds" or "the delay of the first AND gate is greater than the delay of the second AND gate." The state graph constructed with respect to a particular set of delay assumptions rules out some circuit executions which would be allowed under an arbitrary delay model. Hence, formulas in CTL which might not have been true in an arbitrary delay model may be true with respect to particular delay assumptions (because all the counterexample paths are ruled out by the delay assumptions). This technique was applied to a patented asynchronous queue cell in [Browne et al 85]. The authors determined that the circuit did not meet its specifications under the arbitrary element delay

model. However, under the assumption that the input was slower than two of the circuit gates, they showed that the circuit met its temporal logic specifications.

An alternative approach obtains the state diagram by compilation from a specification of the original (synchronous) circuit in a simple programming language-like notation. Browne and Clarke ([Browne et al 86], [Browne & Clarke 86]) use a Pascal-like state machine description language called SML for this purpose. The language includes the standard control structures *if*, *while*, and *loop/exit*. A *cobegin* statement is also provided for simultaneous execution of statements in lock-step. Since SML programs will ultimately be implemented in hardware, the only data types permitted are *boolean* and (bounded) *integer*. The output of the SML compiler is a deterministic Moore Machine that can be automatically implemented as a PLA, PAL, or a ROM. The output can also be analyzed for correctness using the EMC algorithm. In [Browne 86] Browne describes a specialized version of the EMC algorithm that can check Moore machines much more rapidly than the original algorithm.

Another potential area of application is the verification of network communication protocols. The *alternating bit protocol* [Bartlet et al 69] for reliable transmission of messages by a noisy communication channel is a simple example of such an algorithm. By using the CTL model checking procedure it is possible to determine in a few seconds whether this protocol meets its specifications [Clarke et al 86a]. Sifakis at Grenoble [Quielle & Sifakis 81] and Kurshan at Bell Labs [Kurshan 86] have also considered applications involving network protocols. The delay assumptions mentioned above may be useful for describing the *real-time* behavior of such protocols.

8. Conclusion

Although the verification technique described in this paper has already been used to find some nontrivial errors in circuit designs and communications protocols, more research needs to be done before it will become a truly practical debugging tool for use by system designers. One problem is the expressibility of the underlying temporal logic. For circuit specification *timing diagrams* may be more natural to use than temporal logic formulas. Of course, temporal logic is more general since there is no analogue of negation, disjunction, or conjunction for timing diagrams. It may be possible to either systematically translate timing diagrams into temporal logic formulas or check them directly using an algorithm similar to the one used by the model checker. If so, this would simplify the task of specifying a complicated circuit and also allow the designer to be more confident that specifications actually mean what he thinks they mean.

The most important problem, however, is the *state explosion problem*. There are several different strategies for handling this problem. In verifying asynchronous circuits, for example, buggy circuits sometimes result in much larger state graphs than correct circuits. This happens because the activity in the circuit is much more disordered after an error has occurred. One possible solution in this case is to run the program which builds the state graph and the model checker as co-routines, creating states only as they need to be referenced by the model-checker. In [Dill 86] this technique is called *lazy state generation*, by analogy to lazy evaluation in programming language implementations. By using this method, an error could be discovered and reported after constructing only a small part of the entire state graph; this would not only speed up the verification process, it would also make it possible to verify some circuits which could not be verified if the entire graph had to be constructed.

Another approach to the state explosion problem is to exploit the hierarchical structure of complex finite state concurrent systems. If an appropriate subset of CTL is used ([Mishra & Clarke 85], [Clarke et al 86b]), then lower level subcircuits can be simplified by "hiding" some of their internal nodes (more precisely, making it illegal to use them in temporal logic formulas) and merging groups of states that become indistinguishable into single state. Preliminary research in [Mishra & Clarke 85] indicates that by using this technique it may be possible to cut-down dramatically on the number of states that need to be examined.

Finally, special techniques may be appropriate for concurrent systems that are composed of many identical processes. Consider, for example, a distributed mutual exclusion algorithm for processes arranged in a ring network in which mutual exclusion is guaranteed by means of a token that is passed around the ring ([Dijkstra 85], [Kurshan 85], [Martin 85]). A strategy that is often used for debugging such systems is to consider first a reduced system with one or two processes. If it is possible to show that the reduced system is correct and if the individual processes are really identical, then one is tempted to conclude that the entire system will be correct. In [Clarke et al 86b] an attempt is made to provide a solid theoretical basis that will prevent fallacious conclusions in arguments of this type. The authors describe a temporal logic called *Indexed CTL*, or $ICTL$, for specifying networks of identical processes. The logic includes all of CTL with the exception of the nexttime operator; in addition, it permits formulas of the form $\bigwedge_i f(i)$ and $\bigvee_i f(i)$ where $f(i)$ is a formula in which all of the atomic propositions are subscripted by i . A Kripke structure for a family of N identical processes may be obtained as a product of the state graphs of the individual processes. Instances of the same atomic proposition in different processes are distinguished by using the number of the process as a subscript; thus, A_i represents the instance of atomic proposition A associated with process S_i .

Since a closed formula of the new logic cannot contain any atomic propositions with constant index values, it is impossible to refer to a specific process by writing such a formula. Hence, changing the number of processes in a family of identical processes should not effect the truth of a formula in the logic. This intuitive idea is made precise by introducing a new notion of *bisimulation* [Milner 79] between two Kripke structures with the same set of indexed propositions but different sets of index values. It is possible to prove that if two structures correspond in this manner, a closed formula of Indexed CTL^{*} will be true in the initial state of one if and only if it is true in the initial state of the other.

These ideas are illustrated in [Clarke et al 86b] by considering the distributed mutual exclusion algorithm mentioned above. The atomic proposition c_i is true when the i -th process is in its critical region, and the atomic proposition d_i is true when the i -th process is delayed waiting to enter its critical region. A typical requirement for such a system is that a process waiting to enter its critical region will eventually do so. This condition is easily expressed in ICTL^{*} by the formula $\bigwedge_i \text{AG}(d_i \Rightarrow \text{AF}c_i)$. The results of [Clarke et al 86b] can be used to show that exactly the same ICTL^{*} formulas hold in a network with 1000 processes as hold in a network with two processes. The EMC algorithm can be used to check automatically that the above formula holds in networks of size two and conclude that it will also hold in networks of size 1000. At present this methodology has only been partially automated, however. The bisimulation must be established by hand and this generally requires some representation of the larger Kripke structure. Several researchers are attempting to find a way of automating this phase in a manner that avoids building the larger Kripke structure.

Other techniques for avoiding the state explosion problem are being investigated by Kurshan and Wolper. In Kurshan's system [Kurshan 85] this problem is handled by using a homomorphism to collapse a large state machine into a much smaller one while preserving those properties that are important for verification. Since Kurshan does not use temporal logic formulas for specification, he has no analogue of the indexed formulas or of the bisimulation theorem used in [Clarke et al 86b]. Wolper [Wolper 86] considers a logic somewhat like ICTL^{*} for reasoning about programs that are data-independent; however, his indexed variables range over data elements, not over processes. Also, there is no notion of correspondence between structures in his work. Some ultimate limitations on this type of reasoning are discussed in Apt and Kozen [Apt & Kozen 86].

References

- [Apt & Kozen 86] K. Apt and D. Kozen. Limits for Automatic Verification of Finite-State Concurrent Systems. *Inf. Process. Lett.* 22(6):307-309, 1986.
- [Bartlet et al 69] K.A. Bartlet, R.A. Scantlebury, P.T. Wilkinson. A Note on Reliable Full-Duplex Transmission over Half-Duplex Links. *Communications of the ACM* 12(5):260-261, 1969.
- [Ben-Ari et al 83] M. Ben-Ari, Z. Manna, A. Pnueli. The Temporal Logic of Branching Time. *Acta Informatica* (20):207-226, 1983.
- [Bochmann 82] G. V. Bochmann. Hardware Specification with Temporal Logic: An Example. *IEEE Transactions on Computers* C-31(3), March, 1982.
- [Browne 86] M. C. Browne. An Improved Algorithm for the Automatic Verification of Finite State Systems using Temporal Logic. In *Proceedings of the 1986 Conference on Logic in Computer Science.*, pages 260-267. Cambridge, Massachusetts, June, 1986.
- [Browne & Clarke 86] M. C. Browne, E. M. Clarke. SML: A high level language for the design and verification of Finite State Machines. In *IFIP WG 10.2 International Working Conference from HDL Descriptions to Guaranteed Correct Circuit Designs, Grenoble, France.* IFIP, September, 1986.
- [Browne et al 85] M. C. Browne, E. M. Clarke, D. Dill. Checking the Correctness of Sequential Circuits. In *Proceedings of the 1985 International Conference on Computer Design.* IEEE, Port Chester, New York, October, 1985.

[Browne et al 86] M. Browne, E. Clarke, D. Dill, B. Mishra. Automatic Verification of Sequential Circuits using Temporal Logic. *IEEE Transactions on Computers* C-35(12), December, 1986.

[Browne et al 86b] M. C. Browne, E. M. Clarke, and D. Dill. Automatic Circuit Verification Using Temporal Logic: Two New Examples. G.J. Milne and P.A. Subrahmanyam (editors). *Formal Aspects of VLSI Design*. Elsevier Science Publishers (North Holland), 1986b.

[Clarke & Emerson 81] E.M. Clarke, E.A. Emerson. Synthesis of Synchronization Skeletons for Branching Time Temporal Logic. In *Proc. of the Workshop on Logic of Programs*. Springer-Verlag, Yorktown Heights, NY, 1981.

[Clarke et al 86a] E.M. Clarke, E.A. Emerson, A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems* 8(2):244-263, 1986.

[Clarke et al 86b] E. M. Clarke, O. Grumberg, M. C. Browne. Reasoning about Networks with many identical finite-state processes. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing.*, pages 240-248. ACM, August, 1986.

[Dijkstra 85] E. Dijkstra. Invariance and non-determinacy. In C.A.R. Hoare And J.C. Shepherdson (editors), *Mathematical Logic and Programming Languages*, pages 157-163. Prentice-Hall, 1985.

[Dill 86] D. Dill. A Trace Theoretic Approach to Asynchronous Circuit Verification. Workshop on Design and Implementation of Concurrent programs, Groningen, The Netherlands, November 17-21, 1986

[Dill & Clarke 86] David L. Dill and Edmund M. Clarke. Automatic Verification of Asynchronous Circuits using Temporal Logic. *IEE Proceedings* 133, pt. E(5), September, 1986.

[Emerson & Clarke 81] E.A. Emerson and E.M. Clarke. Characterizing Properties of Parallel Programs as Fixpoints. In *Springer Lecture Notes in Computer Science*. Volume 85: *Proc. of the Seventh International Colloquium on Automata, Languages and Programming*. Springer Verlag, 1981.

[Emerson & Halpern 83] E.A. Emerson, J.Y. Halpern. ""Sometimes" and "Not Never" Revisited: On Branching versus Linear Time". In *Proc. 10th ACM Symp. on Principles of Programming Languages*. 1983.

[Emerson & Lei 85] E.A. Emerson, Chin Laung Lei. Modalities for Model Checking: Branching Time Strikes Back. *Twelfth Symposium on Principles of Programming Languages, New Orleans, La.*, January, 1985.

[Francez 86] N. Francez. *Fairness*. Springer Verlag, 1986.

[Gabbay et al 80] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi, The Temporal Analysis of Fairness. *7th ACM Symposium on Principles of Programming Languages*. :164-173, January, 1980.

[Hoare 78] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM* 21(8), August, 1978.

[Hughes & Creswell 77] G.E. Hughes and M.J. Creswell. *An Introduction to Modal Logic*. Methuen and Co., 1977.

- [Kurshan 85] R.P. Kurshan. Modelling Concurrent Processes. In *Proc. of Symposia in Applied Mathematics*. 1985.
- [Kurshan 86] R.P. Kurshan. *Testing Containment of ω -Regular Languages*. Technical Report 1121-861010-33-TM, Bell Laboratories Technical Memorandum, 1986.
- [Lamport 80] L. Lamport. "Sometimes" is Sometimes "Not Never". In *Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 174-185. Association for Computing Machinery, Las Vegas, January, 1980.
- [Lehmann et al 81] D. Lehmann, A. Pnueli, J. Stavi. Impartiality, Justice, and Fairness: The Ethics of Concurrent Termination. *Automata, Languages, and Programming, Springer Verlag LNCS 115*, 1981.
- [Lichtenstein & Pnueli 85] O. Lichtenstein and A. Pnueli. Checking that Finite State Concurrent Programs Satisfy Their Linear Specification. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*. New Orleans, La., January, 1985.
- [Malachi & Owicki 81] Y. Malachi and S. S. Owicki. Temporal Specifications of Self-Timed Systems. In H.T. Kung, Bob Sproull, and G. Steele (editors), *VLSI Systems and Computations*. 1981.
- [Martin 85] A. Martin. The Design of a Self-Timed Circuit for Distributed Mutual Exclusion. In Henry Fuchs (editor), *Proc. 1985 Chapel Hill Conf. on VLSI*, pages 247-260. 1985.
- [Milner 79] R. Milner. *Lecture Notes in Computer Science*. Volume 92: *A Calculus of Communicating Systems*. Springer-Verlag, 1979.

- [Mishra & Clarke 85] B. Mishra, E.M. Clarke. Hierarchical Verification of Asynchronous Circuits using Temporal Logic. *Theoretical Computer Science* 38:269-291, 1985.
- [Owicki & Lamport 82] S. Owicki, L. Lamport. Proving Liveness Properties of Concurrent Programs. *ACM Transactions on Programming Languages and Systems* 4(3):455-495, July, 1982.
- [Pneuli 77] A. Pneuli. The Temporal Semantics of Concurrent Programs. In *18th Symposium on Foundations of Computer Science*. 1977.
- [Quielle & Sifakis 81] J.P. Quielle, J. Sifakis. "Specification and Verification of Concurrent Systems in CESAR". In *Proc. of the Fifth International Symposium in Programming*. 1981.
- [Quielle & Sifakis 82] J.P. Quielle, J. Sifakis. Fairness and Related Properties in Transition Systems. *IMAG* (292), March, 1982.
- [Seitz 80] C. Seitz. System Timing. *Introduction to VLSI Systems (C. Mead and L. Conway)*. Reading, MA, Addison-Wesley, 1980.
- [Sistla & Clarke 86] A.P. Sistla, E.M. Clarke. Complexity of Propositional Temporal Logics. *Journal of the Association for Computing Machinery* 32(3):733-749, July, 1986.
- [Vardi & Wolper 86] M. Vardi and P. Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *Proceedings of the Conference on Logic in Computer Science*. Boston, Mass., June, 1986.
- [Wolper 86] P. Wolper. Expressing Interesting Properties of Programs in Propositional Temporal Logic. In *Thirteenth ACM Symposium on Principles of Programming Languages*. 1986.

END

FILMED

MARCH, 19 88

DTIC